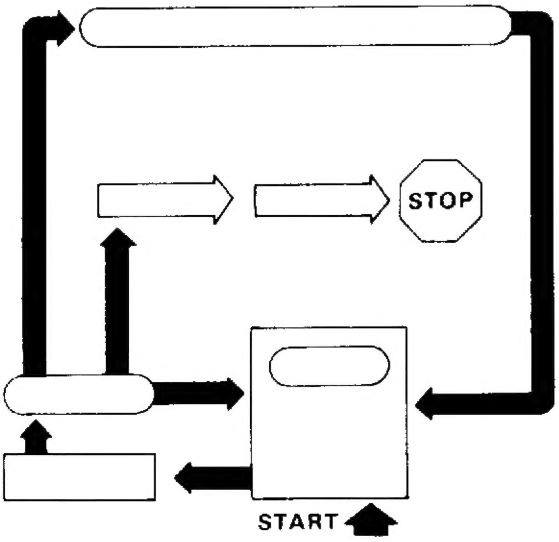
An instructional manual for

CARDIAC

A cardboard illustrative aid to computation

David Hagelbarger
Saul Fingerman
Bell Telephone Laboratories



An instructional manual for

CARDIAC

A cardboard illustrative aid to computation

by David Hagelbarger
Saul Fingerman
Bell Telephone Laboratories

Cartoon illustrations by A. Barthelson

Bell System Educational Aid
Developed by Bell Telephone Laboratories

Third Printing, April 1970

Table of Contents

Preface

You may be surprised that CARDIAC, which is about computers, has been developed and made available by the Bell System. It is true that the Bell System's relationship with computers is not obvious but nevertheless it is very substantial.

To begin with, the Bell System is the nation's largest user of computers (with the exception of the federal government). Hundreds of computers are used for billing, record keeping and other internal operations. The Bell System also offers a number of services for interconnecting computers, or connecting with them.

In addition, the entire Bell System telephone network has often been compared to a gigantic computer: Digital information in the form of a called number is pulsed into a central office, switching equipment possessing computer-like features then solves the problem of establishing a connection between the called and calling telephones.

Out of the research and development that made this network possible, has blossomed much of the basic technology of modern computers.

But it is at Bell Telephone Laboratories—the research and development unit of the Bell System—that the System's most extensive involvement with computers is to be found.

The first electrical digital computers were conceived at Bell Laboratories (as well as at Harvard University, in an entirely independent effort), shortly before World War II. The inventor was George Stibitz, who later went on to develop several other computers that remained in productive service throughout the war. These were all relay machines—primitive by comparison to the incredibly efficient electronic computers that have become so much a part of contemporary life.

What made efficiency possible was the transistor. Invented at Bell Laboratories, the transistor not only cut size and power requirements, but also provided the speed and reliability that makes it possible for computers to perform millions of operations without errors.

Within a decade of its inception, the transistor was proved out with TRADIC, an airborne computer built by Bell Laboratories for the military. Since then, direct descendants of TRADIC have played essential roles in other military programs, such as the SAGE communications network and the very complex Nike Zeus and Nike-X antiballistic-missile projects.

The computer is playing an increasingly important part in *all* areas of research at Bell Laboratories. Today, about 30 per cent of the technical personnel there spend more than half their time programming computers.

Currently, a Bell Laboratories task force is developing BIS (Business Information System) to supply Bell System management with the kind of up-to-the-minute information needed to reduce operating costs and provide better customer service. BIS will use new third generation computers—high-speed, on-line, real-time random-access machines with mass information storage and retrieval capabilities.

It is no exaggeration to say the story of Bell Laboratories and computers is a significant one. Information theory, error detection and correction codes, electronic switching, programs for visual computer displays such as BEFLIX, as well as for design, simulation and modeling—these and many others are only highlights in the long story. And as a by-product of this story, CARDIAC was developed, which we hope will help you to understand computers.

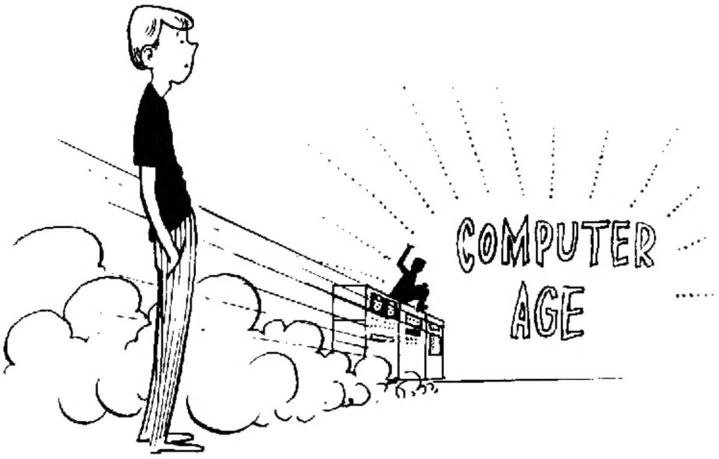
C. I. R.

SECTION 1.

WHAT CARDIAC IS ...AND ISN'T

CARDIAC is an acronym for CARDboard Illustrative Aid to Computation. The key word here is "illustrative." It means that CARDIAC *illustrates* the operation of a computer without actually *being* a computer. In fact, it is not even a practical *aid* to computing. On the other hand, it is a very practical aid to *understanding* computers and computer programming.

You'll need this kind of understanding to keep up with the Computer Age you are about to enter. These are fast-moving times, and those who make no effort to understand computers may very well get left behind.

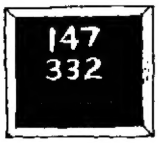


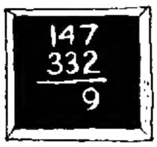
SECTION 2.

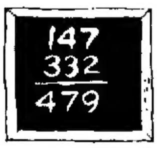
BASIC UNITS OF A SIMPLE COMPUTER

Before we get into computers or CARDIAC, it might be a good idea to see what's actually involved in computing. Most of us can compute without too much trouble. In fact, we're so good at it, we add, subtract, multiply, and divide without giving any thought to the mechanics of what we're doing. But, to understand computers, we'll temporarily have to discard these automatic skills and take a closer, step-by-step look at what we do.

Let's go back several years to one of your earliest arithmetic classes. The teacher has just called you to the blackboard to solve a problem: Add 147 to 332.





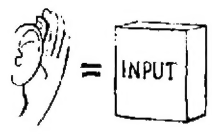


After clearing your mind of everything but your newly learned procedures for addition, you write the two numbers on the board. You write them in column form, as you have been taught to do—one below the other. Then, you draw a line beneath them and begin adding the right-hand column. "Seven plus two equal nine," you say, and dutifully write a nine below the line. "Four plus three equal seven," and you write a seven below the line. "One plus three equal four," and you write that down, too. Are you finished? Not quite. You know the teacher is waiting to hear the results so you loudly call out, "The answer is 479." *Now*, you're finished.

Simple Computer Block Diagram

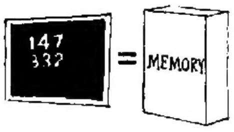
Let's see exactly what you did to get that answer. At the same time, we'll begin diagramming a simple computer that can do the same things.

Input



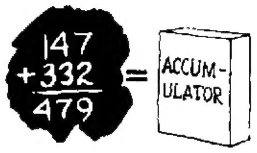
First of all, you *listened*. As soon as the teacher called your name, you began taking mental notes of everything she said, paying particular attention to three words: The words were "add" (an instruction), "147" (data), and "332" (data). Our computer will also need a similar input device to receive instructions and data.

Memory



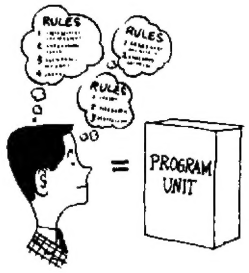
After receiving the data, you need something convenient in which to store, or *remember* it, so you wrote the numbers on the blackboard. Our computer will also need a memory.

Accumulator



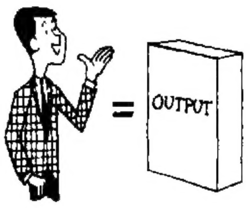
Next, you proceeded to add the two data numbers column by column, writing the partial results as you went, until the final sum had accumulated below the line. In short, you carried out the necessary arithmetic operations. Our computer must also have an arithmetic unit. Since the results of every arithmetic operation will accumulate within this unit, we'll call it an *accumulator*.

Program



Recall now, that when you stored your data in the blackboard memory, you didn't simply write the two numbers in random fashion. Instead, you wrote them neatly one below the other in column form—in the prescribed fashion for addition. Recall also, that in adding the numbers, you began with the right-hand column, added it, moved on to the next column, and so on. You performed each step according to a rigidly prescribed set of instructions that you had previously learned. This set of instructions, which guided you through the entire problem, we call the *program*. Our still-to-be-completed computer would be useless without one.

Output



The last thing you did before erasing the board and returning to your seat was to call out the answer to your teacher. Our computer will also have to make its answers available to us in some recognizable form. For this operation, we will need an *output* device.

Our block diagram is now complete enough to warrant a name. Since it's a fairly simple computer, we'll call it SIMCO, for short.

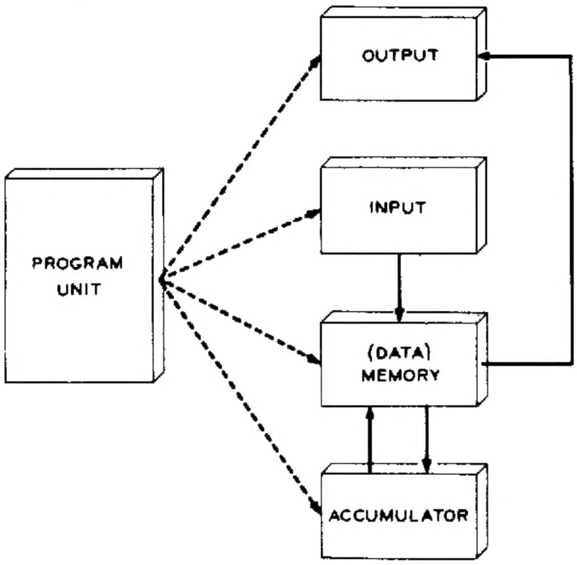


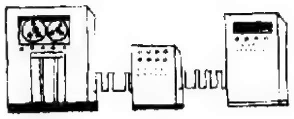
Fig. No. 1. Block diagram for SIMCO.

SECTION 3.

COMMUNICATING WITH COMPUTERS

Input

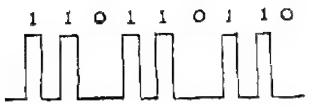
The next section will introduce two words that computer people use in a very special sense. The words are READ and PRINT, and their use involves the input and output sections of a computer. To explain their special meanings, it will be helpful to turn momentarily to the devices actually used by real computers.



Science fiction writers to the contrary, computers are only machines—no more and no less. The various elements of a computer must communicate with each other in a language machines can "understand"—specifically, an electronic language. By and large, the "alphabet" of this language consists of electrical pulses, and the "words" are made up of sequences of such pulses arranged according to standard codes.

If we designate the absence of a pulse as *zero* and the presence of a pulse as *one*, we can encode almost any kind of information we want into groups of *one's* and *zero's*.

The input section of a computer is simply a device for entering such *one's* and *zero's* into a computer in the form of pulses. One of



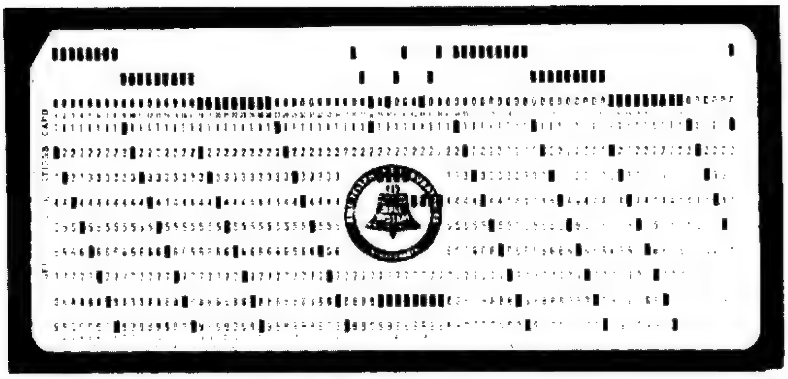


Fig. No. 2. Punched card.

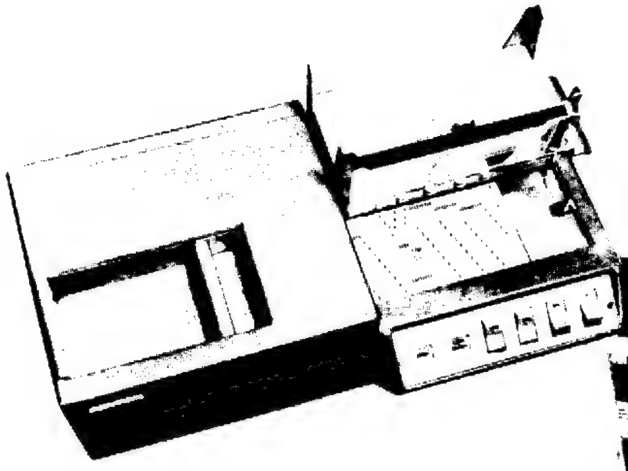


Fig. No. 3. Punched card reader.

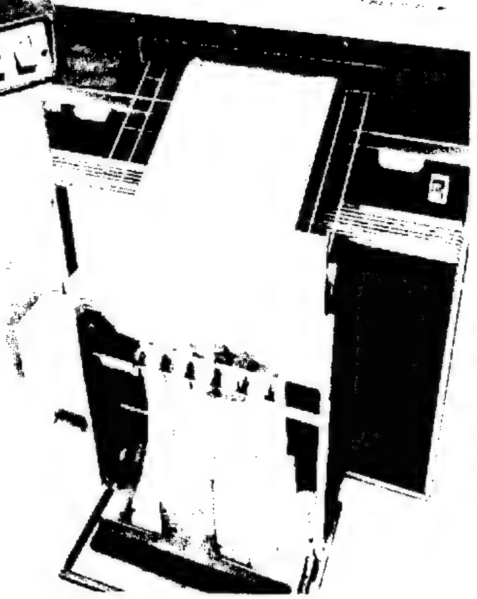


Fig. No. 4. High speed printer.

the most common input devices is the punched card reader shown in Fig. 3. Fig. 2 shows a typical punched card.

Tiny metal feelers in the card reader sense the presence (or absence) of holes in punched cards. The feelers act as switches that relay pulses (or no pulses) to the computer's memory section. Thus, each combination of holes and no-holes represents a letter or a number.

Hereafter, when we order our computer to *read* something, we will be telling it to take a piece of information from the input for storage in some specified location of the memory.

Output Devices

Output devices are the computer's means of communicating with us. Not surprisingly, their operation is pretty much the reverse of input devices. They take pulsed information from the memory section and convert it into some form we can understand.

One of the most common forms of output devices is the high-speed printer shown in Fig. 4. It can convert into print pulsed information from a computer's memory at the rate of hundreds of lines per minute. Hereafter, when we command our imaginary computer *to print* something, we will be telling it to take some information from a specified location in the memory and print it out.

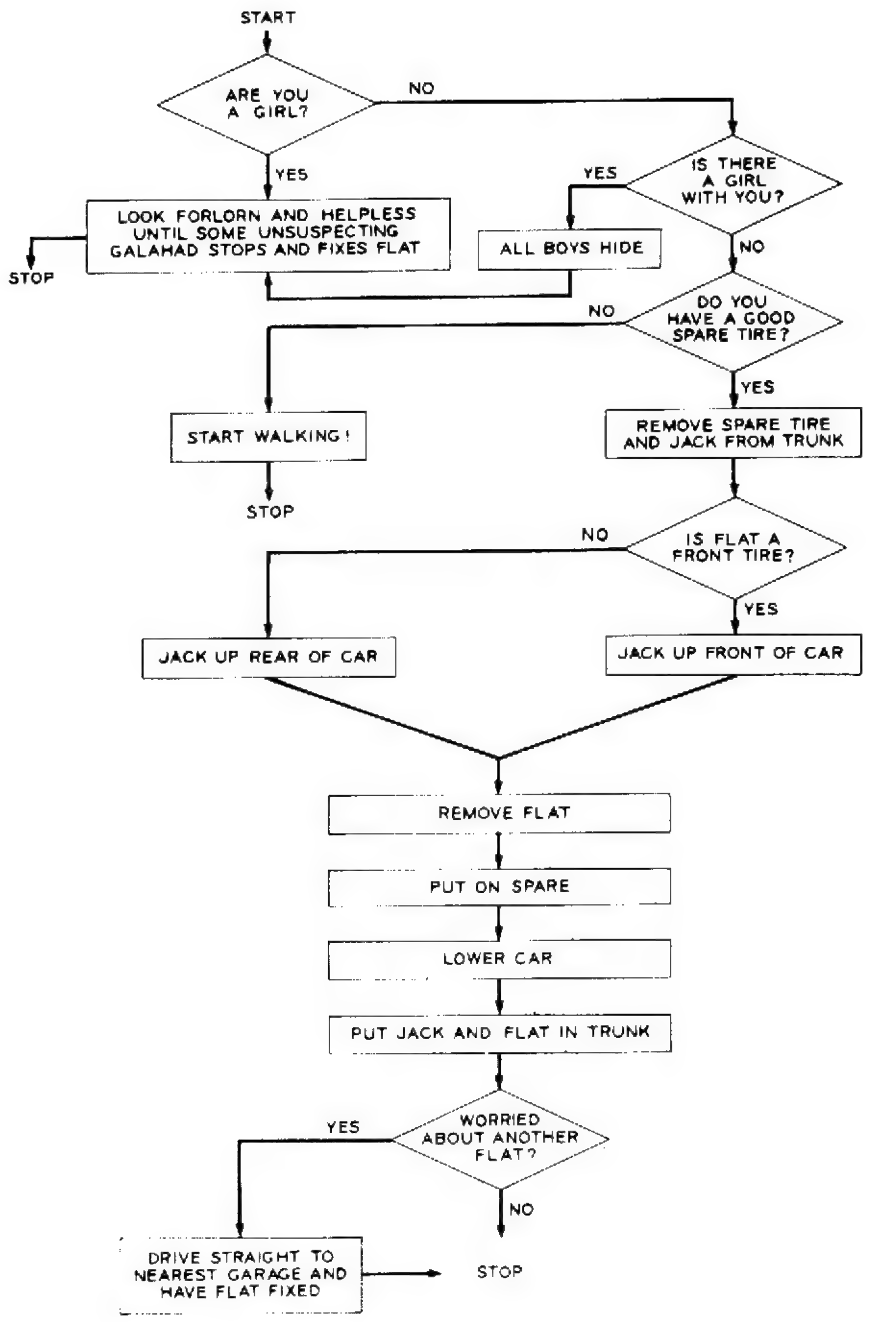
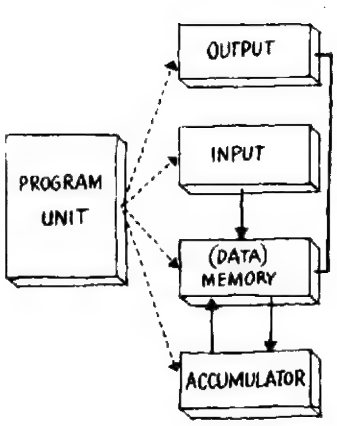

START
ARE YOU A GIRL?
NO
YES
IS THERE A GIRL WITH YOU?
YES
LOOK FORLORN AND HELPLESS UNTIL SOME UNSUSPECTING GALAHAD STOPS AND FIXES FLAT
STOP
ALL BOYS HIDE
NO
DO YOU HAVE A GOOD SPARE TIRE?
NO
YES
START WALKING!
STOP
REMOVE SPARE TIRE AND JACK FROM TRUNK
IS FLAT A FRONT TIRE?
NO
YES
JACK UP REAR OF CAR
JACK UP FRONT OF CAR
REMOVE FLAT
PUT ON SPARE
LOWER CAR
PUT JACK AND FLAT IN TRUNK
WORRIED ABOUT ANOTHER FLAT?
YES
NO
DRIVE STRAIGHT TO NEAREST GARAGE AND HAVE FLAT FIXED
STOP

Fig. No. 5. Flow chart for repairing a flat tire.

SECTION 4.

FLOW CHARTS



The block diagram we have constructed contains the basic elements of a simple computer. However, it doesn't tell us much about the interactions occurring between these units. Now, we're ready to begin the important business of studying exactly what these interactions are.

One way to begin visualizing the internal dynamics of a computer is to draw a *flow chart*—a step-by-step diagram of all operations involved in the solution of a particular problem. Flow charts can be drawn for nearly every kind of activity imaginable, including changing a flat tire—as shown on the facing page. A flow chart for a student adding two numbers at the blackboard is shown below:

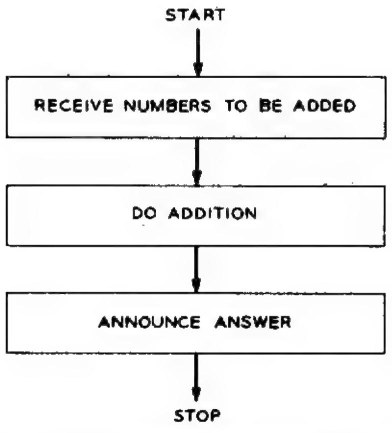


Fig. No. 6. Flow chart for blackboard addition.

This flow chart may seem fairly obvious; but, as problems and procedures become more complicated, flow charts become more helpful as an intermediate step between analyzing a problem and programming a computer to solve it.

For example, even the flow chart for adding two numbers becomes somewhat more involved when it is drawn up for SIMCO.

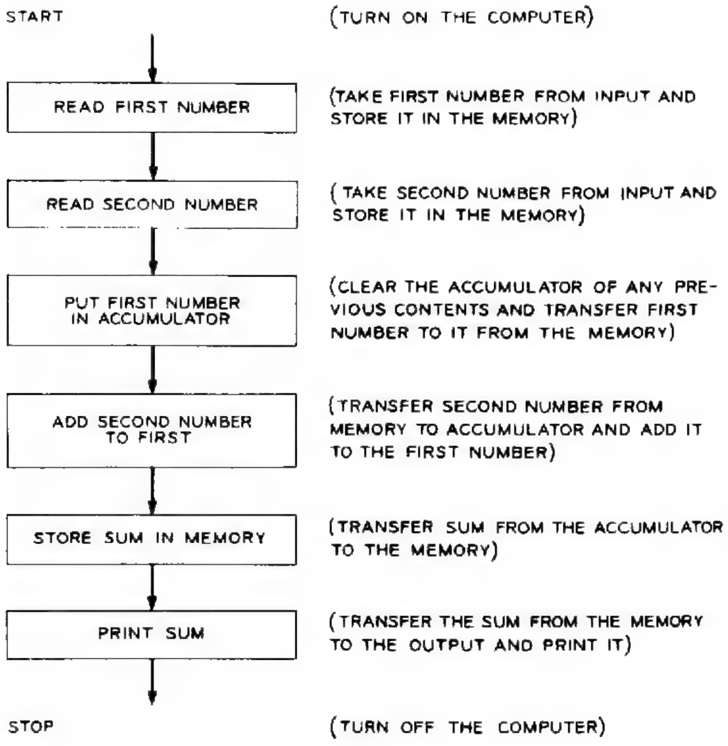

START
(TURN ON THE COMPUTER)
READ FIRST NUMBER
(TAKE FIRST NUMBER FROM INPUT AND STORE IT IN THE MEMORY)
READ SECOND NUMBER
(TAKE SECOND NUMBER FROM INPUT AND STORE IT IN THE MEMORY)
PUT FIRST NUMBER IN ACCUMULATOR
(CLEAR THE ACCUMULATOR OF ANY PREVIOUS CONTENTS AND TRANSFER FIRST NUMBER TO IT FROM THE MEMORY)
ADD SECOND NUMBER TO FIRST
(TRANSFER SECOND NUMBER FROM MEMORY TO ACCUMULATOR AND ADD IT TO THE FIRST NUMBER)
STORE SUM IN MEMORY
(TRANSFER SUM FROM THE ACCUMULATOR TO THE MEMORY)
PRINT SUM
(TRANSFER THE SUM FROM THE MEMORY TO THE OUTPUT AND PRINT IT)
STOP
(TURN OFF THE COMPUTER)

Fig. No. 7. Flow chart for addition with SIMCO.

Action Diagrams

As shown on page 9, the action indicated in each step of our flow chart can be illustrated by a series of block diagrams. Data flow is shown by solid lines. Dashed lines indicate flow of controlling pulses.

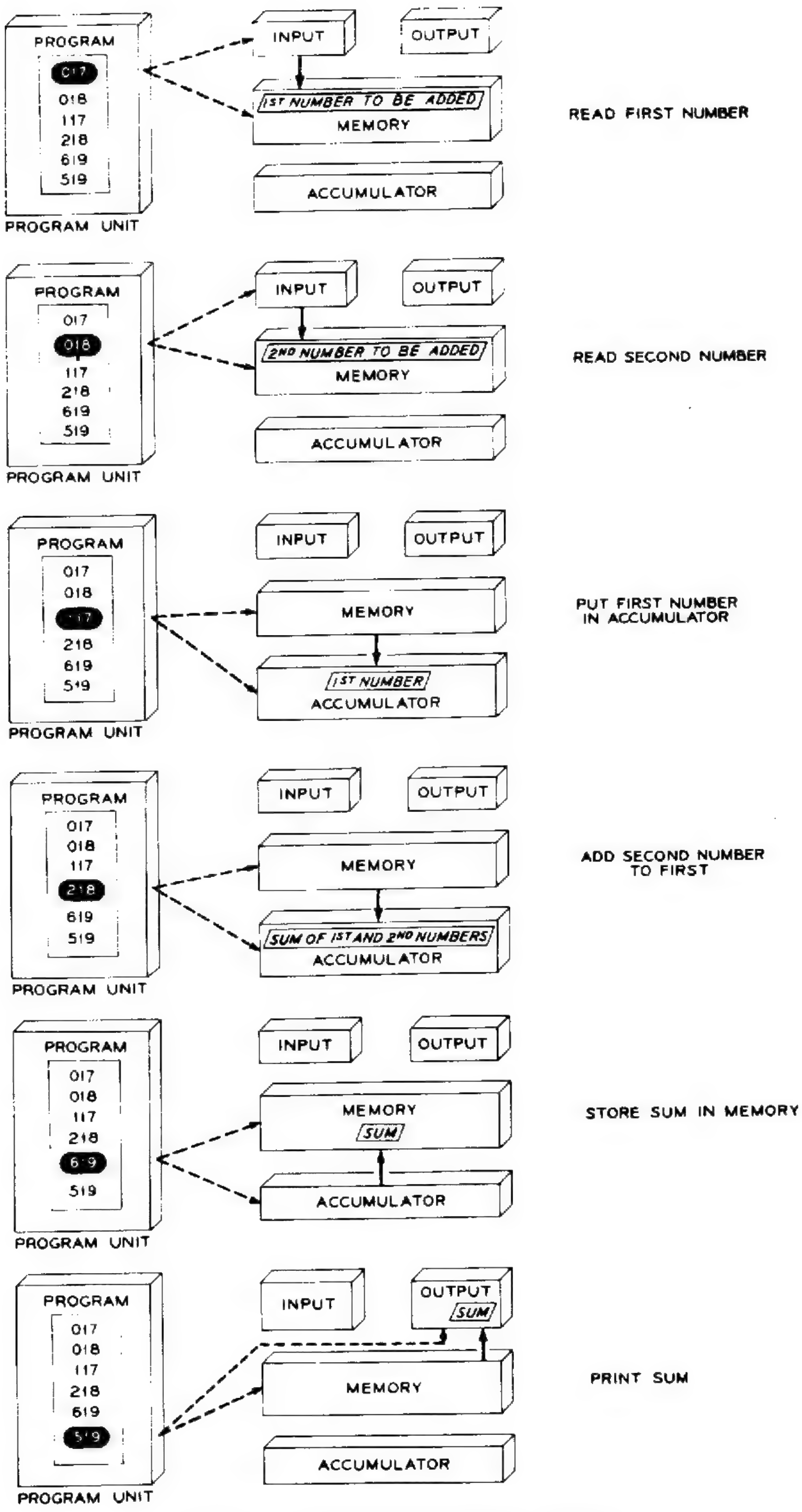
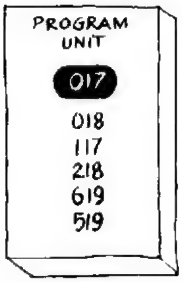


Fig. No. 8. SIMCO adding two numbers.

SECTION 5.

INSTRUCTIONS, DATA, AND ADDRESSES



The mysterious looking numbers in the program units in Fig. 8 are machine-language abbreviations of the verbal instructions contained in the flow chart. Their meaning will be made clear as we go. For the moment, it's necessary to understand only that they are perfectly clear to the computer,* and that the program unit supplies them in the correct sequence.

Operational Codes

OP CODE
017
018

Each instruction, or "word," in the program unit consists of three digits. The first digit of each program word is the *operational code*—a command to the computer to perform a specific operation such as "read," "print," "add," or "subtract." Incredible as it seems, only ten such operations are needed to solve almost any problem for which a precise method of solution can be stated!

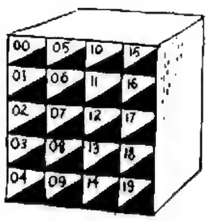
The ten operational codes for SIMCO are:

0 = Input	5 = Output
1 = Clear and Add	6 = Store
2 = Add	7 = Subtract
3 = Test Accumulator Contents	8 = Unconditional Jump
4 = Shift	9 = Halt and Reset

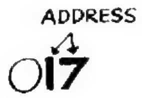
These are the same operational codes used by CARDIAC. As you'll see, there are many other functional similarities, including CARDIAC's ability to solve any problem that can be solved by SIMCO.

*It is possible to conceive of a computer capable of responding to verbal instructions like the ones in our flow chart. For example, an input device could be designed to understand the command, "Take the first number from the memory and add it to the number in the accumulator." However, there are literally thousands of word combinations for saying the same thing. Designing a computer that could generalize a single, unambiguous meaning from all these combinations is still possible only in the realm of science fiction.

Memory and Addresses

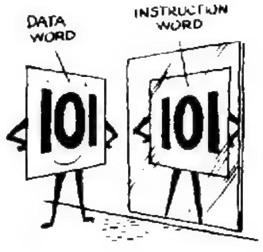


All the time it is working, a computer is constantly shuttling data to and from its memory unit. Obviously, each item stored in the memory must be kept separate and distinct from every other item. What's more, each item must be stored so that it can be instantly retrieved when needed. This kind of instantaneous and foolproof *access* to the memory is possible only if it is physically divided into a number of distinctly identifiable locations. (Think of these as electronic pigeon holes.) Each location has its own number, or *address*. Large computers typically have as many as fifteen thousand addresses. For practical reasons, SIMCO's memory (and CARDIAC's) consists of 100 addresses, numbered 00 through 99. The last two digits of each program word correspond to one of these addresses. Thus, our computer works with 3-digit words: one digit for the operational code, and two for a memory address.



Instruction Words and Data Words Are Look-Alikes

The remarkable thing is that all instructions can be given in this same, invariable format of a 3-digit word.

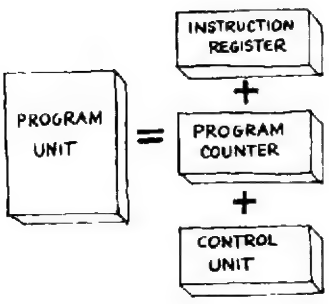


But, if our instructions words are made up of three digits, what do our data words look like? Answer: They look exactly the same. Data words—the material being processed—are also made up of three digits. It would probably be more accurate to say their length is *limited* to three digits, since this is the maximum capacity of each memory cell. In any case, the point is that both types of words look the same.

Before we can answer the intriguing question of how the computer tells them apart, we must first recognize that this similarity is an enormous advantage. It means, first of all, that both kinds of words can be processed by the same hardware. They can be fed into the same input devices, operated on in the same accumulator, and stored in the same memory. This not only makes for greater economy, but also means that, as a computer is proceeding through a problem, *it can process its own* instructions. Computers functioning this way are known as stored-program computers. It is the ability to store and revise their own program that gives stored-program computers the appearance of being almost completely automatic in their operation.

SECTION 6.

CONVERTING SIMCO TO THE STORED-PROGRAM MODE



To make SIMCO a stored-program computer, we have to replace its inefficient program unit with three new devices—an *instruction register*, a *program counter*, and a *control unit*. The addition of these units makes SIMCO the equal of any real computer.

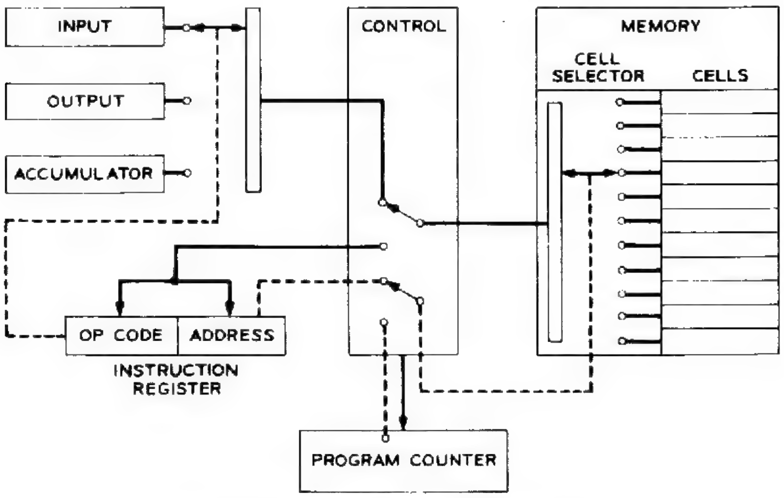


Fig. No. 9. Stored-program computer.

The Instruction Register

The function of the instruction register is to store each instruction word during the time that particular instruction is being executed. Unlike SIMCO's cumbersome program unit, which had to store an entire program, the instruction register needs to store only *one* instruction at a time. Once this instruction is executed, the instruction register is fed a new word.

The Program Counter

Among its many other functions, SIMCO's program unit had to feed out instructions in the proper sequence. To do this, it had to keep tabs of exactly *where it was* in this sequence.

Like a man following a long list of written instructions, it kept its place by "moving its finger" to the next instruction before executing the previous one. The program counter contains the memory address from which the current instruction was fetched. Before the instruction is executed, 1 is added to this address. Since instructions are stored sequentially in memory, increasing the address by 1 automatically provides the correct address for the next instruction.

The program counter also requires another capability: Although instructions are stored sequentially in the memory, a computer often has to repeat an earlier instruction or even jump ahead to another, not in sequence. Instructions that elicit this response are known as *jump instructions*. Their purpose will be fully explained later. Till then, it's enough to know that the program counter must sometimes be able to change its count (and, hence, the address of the next instruction) by more than one.

The Control Unit

The control unit controls the operation of the instruction register and program counter in relation to all the computer's other units. It is a connecting, or *switching*, device; something like a telephone operator who sets up connections in response to signals on her switchboard.

Specifically, the control unit:

1. Increases the number in the program counter by one, thereby changing that number to the address of the next instruction.

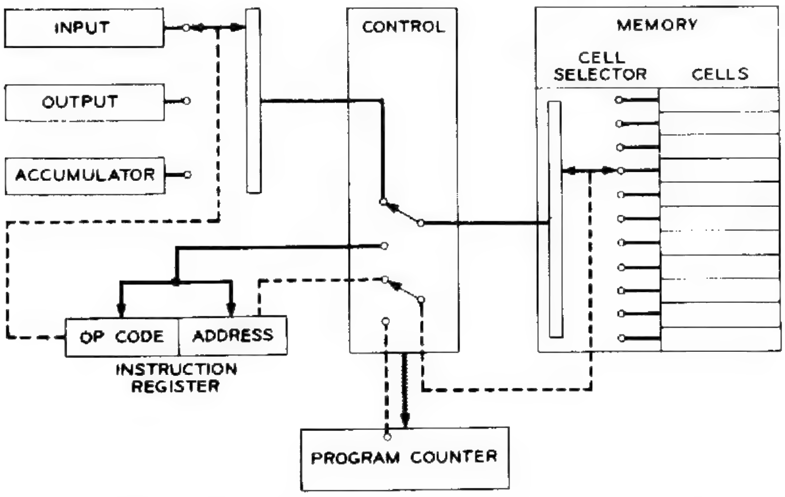


Fig. No. 10. Control adding 1 to the program counter.

2. Uses the number in the program counter as the address from which to fetch the next instruction word to the instruction register. The program counter directs the memory cell selector to the proper cell.

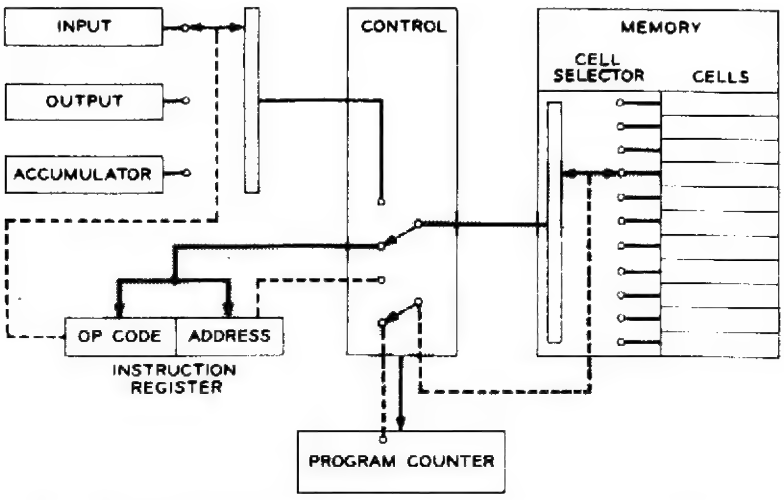


Fig. No. 11. Control fetching a word to the instruction register.

3. Activates the instruction register to execute the current instruction. The instruction shown being executed is causing a word to be read from the input into the memory.

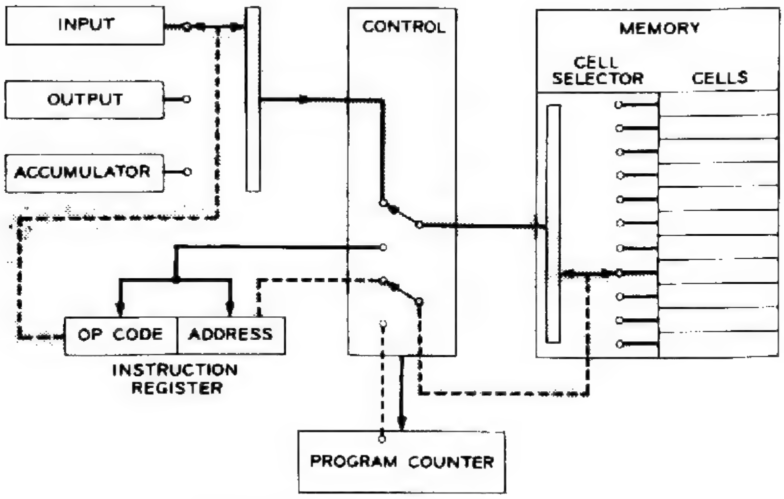
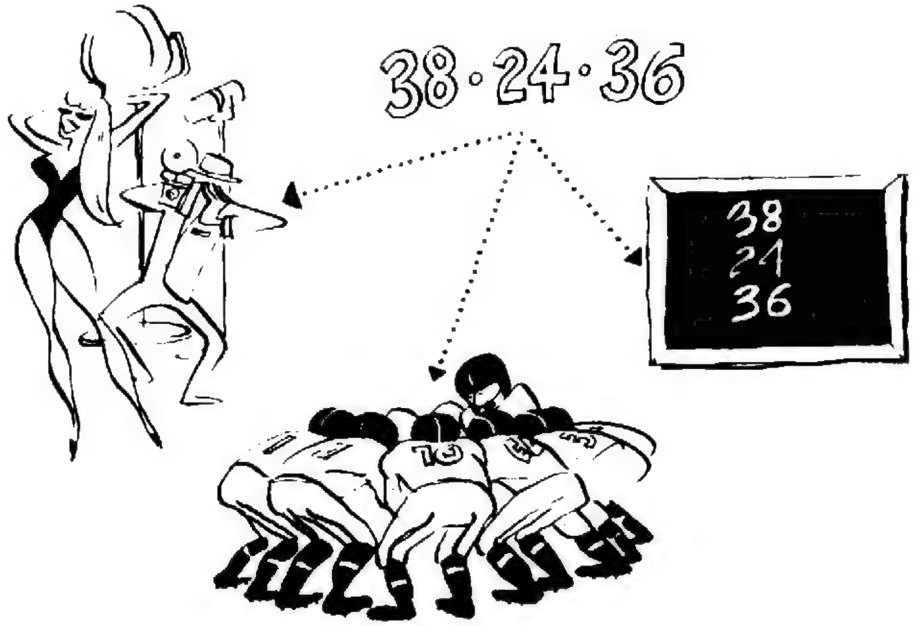


Fig. No. 12. Instruction register reading a word into the memory.

How the Computer Tells Instruction Words from Data Words

We are now ready to look into the question of how the various units of a computer can distinguish between instruction words and data words. From a computer's point of view, the solution is simple: The use and meaning of a word depend entirely on which unit of the computer it happens to be in.

This is analogous to the way a set of numbers, such as 38-24-36, can be variously interpreted depending on where they are used. In a Hollywood movie studio, they would probably indicate the dimensions of a curvaceous actress (data). In a football stadium, they would most likely be signals for the next play (instructions). In an elementary arithmetic classroom, they could simply be three numbers to be added (data).



Similarly, the meaning of a 3-digit word in our computer depends on *where* in the computer it happens to be. For example, "017" in the instruction register will mean "Take the word now appearing in the input and store it in memory cell 17."

In the accumulator, 017 will be treated purely as the data number seventeen and will be added or subtracted to or from any other number already in the accumulator.

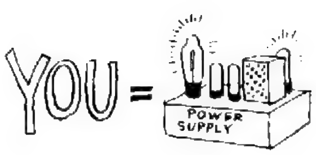
As for the memory unit, 017 can be either an instruction word or a data word. Its use will depend on what other unit the program calls for it to be fetched to. If it is a data word and the program mistakenly calls for it to be fetched to the instruction register, it will be treated as an instruction, and the computer will go slightly insane. This is an all too common error of programmers, and computers have been known to do pretty strange things because of it. Fortunately, the results of such mistakes are usually so outlandish, computer operators can soon see that something has gone wrong and remove the program for correction.

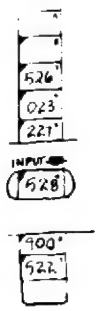
SECTION 7

INTRODUCING CARDIAC

Now that we have assembled a complete block diagram of a computer, we are ready to correlate its elements with the analogous elements of CARDIAC. Some of these analogies are fairly obvious and will require little comment. Others, less obvious, will be explained in detail.

NOTE: One unit has been deliberately omitted from our block diagram. It is the power supply, or energy source. It was omitted because, when working with CARDIAC, *you* will be the energy source. *You* will operate the slides and transfer data from one section of CARDIAC to another. You will even do the arithmetic that must be done in the accumulator. This in no way detracts from CARDIAC's power as a learning tool. Remember, you are not working with CARDIAC to learn arithmetic, but to learn how a computer operates.





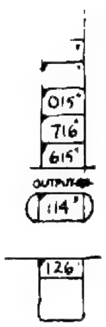
Input

As was previously mentioned, a computer's input devices are the means by which data and a program of instructions are entered into the computer for storage in the memory.

Since one of the most common input devices is the punched card reader, CARDIAC's input has been made in the shape of a strip of punched cards attached end to end.

After pencilling our program and data on the strip, it will be inserted in the input slot. Card number 1 should appear just below the arrow.

During the course of a problem, instructions directing the flow of input information will appear in the instruction decoder window.

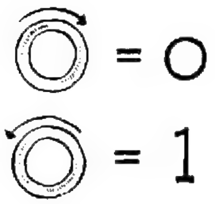


Output

CARDIAC's output looks and functions like its input. The strip of cards is inserted in the output slot with card number 1 appearing under the window. During its operation, CARDIAC's control section will direct the flow of any output data generated to the cards.

Memory

A computer "remembers" things magnetically. Its primary memory is usually made up of thousands upon thousands of tiny ferrite cores, each capable of storing one bit, or *binary digit*, of information. Clockwise magnetization of a core indicates a binary zero, while counterclockwise magnetization indicates a binary one. Information pulsed to these cores from other circuits change their polarity one way or the other.



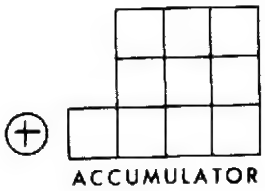
The core memories of very large computers can store as many as 192,000 words, each consisting of thirty-six or more binary digits.

To keep things simple, CARDIAC works with decimal, rather than binary, digits. Also, its memory is considerably smaller. It can store only 100 three-digit words, and these are entered by pencil and retrieved visually.

Accumulator

The accumulator is a computer's arithmetic unit. In it, numbers are added, subtracted, or subjected to operations such as shifting of digits to the left or right.

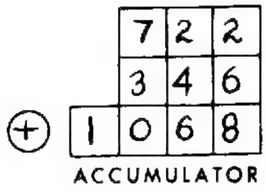
Numbers in the accumulator can also be tested for their sign—negative or positive.



CARDIAC's accumulator fulfills the same purposes. However, *you* will function as its electronics by executing any arithmetic called for by the instruction register. You will also set the accumulator-sign slide so that the correct sign appears in the circular window.

Similarly, when entering numbers on the input or output cards or in the memory, you will have to indicate if they are negative. Unsigned numbers imply that they are positive.

The accumulator proper consists of only the bottom row of squares; the two upper rows serve only as a scratchpad for addition and subtraction.

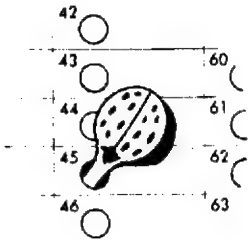


Since CARDIAC's memory can store only 3-digit numbers, you may be puzzled by the inclusion of an extra square in the accumulator. It is there to handle the overflow that will result when two 3-digit numbers whose sum exceeds 999 are added.

Program Counter

The program counter keeps track of which step of a program a computer should execute next. It is actually an electronic counter whose count represents the address of the memory cell from which the next instruction must be fetched.

Because the instruction register sometimes calls for an instruction out of sequence, the program counter must be resettable to any number dictated by the instruction register. Usually, however, it simply increases its count by one, automatically setting itself to the address of the next memory cell.



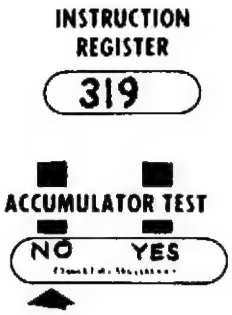
CARDIAC's program counter is simply a marker in the shape of a lady bug. During the course of a program, it is manually moved from one memory cell to another. If used properly, it will mark your place as accurately as an electronic program counter and, probably, require less maintenance.

Instruction Register

If you were to add two numbers on an ordinary desk calculator, you would most likely go through each of the following steps:

1. Read the instruction ("Add number A to number B").
2. Look at number A.
3. Punch the keys for number A.
4. Look at number B.
5. Punch the keys for number B.
6. Punch the "total" key.

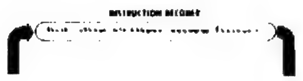
A computer's instruction register accomplishes the equivalent of all this key punching by first storing an instruction and then pulsing the correct circuits to execute it. *Which* circuits the pulses go to is determined by the operation code of the instruction word being pulsed. This is analogous to what happens when you dial the ten digits of a long distance telephone number. The first three digits (the area code) activate equipment that routes the call to the proper city. The next three digits select the correct exchange in that city. Then, the remaining four digits operate the necessary switching equipment to connect you with the individual telephone you are calling.



CARDIAC's instruction register consists of the op-code and address slides plus the three windows that display the material printed on them.

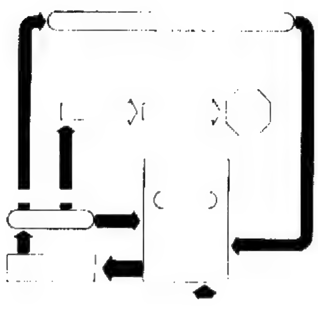
The window labeled "instruction register" allows us to look into the register to see the instruction word stored in it.

The "accumulator test" window is used to test the sign of a number in the accumulator. It also tests the input to see if all cards have been read into the memory.



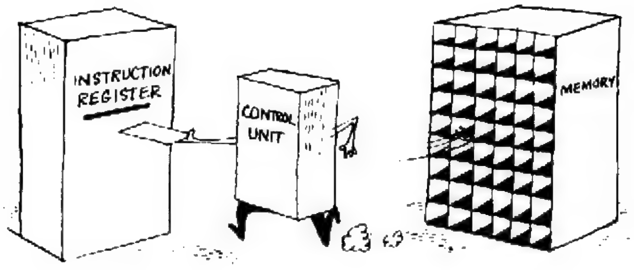
The instruction-decoder window, in a sense, generates the pulses that activate the correct circuits to execute an instruction. Since *you* are substituting for these circuits, the instructions are written in English rather than in pulses. You can think of this window as decoding the pulses for you.

Sequencing



In a real computer, the *sequence* of instruction pulses generated by the instruction register is all important. In CARDIAC, too, correct sequencing is all important. The flow chart path indicated by the arrows must be rigidly followed from the instruction-register window to the accumulator-test window to the instruction-decoder window and back to the instruction-register window.

This flow represents the innermost cycle of a computer's operation.



Control

A computer's control unit also follows an invariable three-part cycle.

First of all, it fetches an instruction from the memory to the instruction register.

Next, it increments the number in the program counter, raising that number to the address of the next instruction.

Finally, it triggers the instruction register into executing the previously fetched instruction. While the instruction register is going through its cycle, the control unit remains quiescent. It assumes control again only after the instruction register has completed an instruction.

You will serve as CARDIAC's control unit by visually following its internal flow chart. While doing so, you will perform all of the operations described above.

As you go through each cycle, you should occasionally pause to remember that computers go through the same cycle about a million times faster.

SECTION 8.

THE FIRST PROGRAM

Now that we have examined CARDIAC's different sections, we are ready to run through our first program. It is an extremely simple program for adding two numbers. What's more, we have written it for you. Its purpose is not to impress you with CARDIAC's power as a computer, but, rather, to familiarize you with its operation. Later programs won't be so simple—particularly when *you* try your hand at writing them.

In going through this program, please don't anticipate, or jump ahead. Skipping even a single step can cause calamitous results. *You* may find the procedure tedious, but computers do not. They do the same thing billions of times a day, tirelessly and without even a twinge of boredom. This is their power. Eventually, you will come to appreciate it as much as the scientists and mathematicians who used to spend days, and even months in computational drudgery.

Because our program consists of only seven instruction words, we could simply write them on a single line like this: 034, 035, 134, 235, 636, 536, 900. However, a longer program, lumped together like that, would be a mess. It's much better to use the tabular format shown below.

The first column lists the addresses of the memory cells into which the program words are to be loaded. The second column is the program proper and will become the contents of these cells. The third column contains explanatory comments.

Program No. 1: Adding Number "A" to Number "B" to Produce Sum "S"

ADDRESS	CONTENTS	COMMENTS
17	034	Read "A".
18	035	Read "B".
19	134	Clear accumulator and add "A".
20	235	Add "B" ("S" is now in accumulator).
21	636	Store "S".
22	536	Print "S".
23	900	Halt and reset.

Directions

1. Using a soft (2B) pencil, lightly write the program words in the indicated memory cells.
2. Write the two numbers to be added on the first and second cards of an input strip. (Use any two numbers whose sum doesn't exceed 999). Insert the strip in the input slot with card number 1 appearing under the arrow.
3. Put the bug (program counter) in the punched hole of memory cell number 17.
4. Insert a blank card strip into the output slot with card number 1 appearing in the window.
5. Start.

More on OP Codes

If you followed the program carefully, CARDIAC will have produced the correct sum on an output card.

Note that the last instruction "900" not only halted the machine, but also reset the program counter to zero.

Six different operational codes were used in this program. They are listed below along with their mnemonic abbreviations and explanations. The four remaining op codes will be explained as they are introduced.

OP CODE	ABBREVIATION	OPERATION
0 _ _	INP	Read input card into cell _ _.
1 _ _	CLA	Clear accumulator and add into it the contents of cell _ _.
2 _ _	ADD	Add contents of cell _ _ into accumulator.
5 _ _	OUT	Print contents of cell _ _ on output card.
6 _ _	STO	Store contents of accumulator in cell _ _.
9 _ _	HRS	Halt machine and reset program counter to _ _.

Where Should Programs Begin?

You may have wondered why our program began in memory cell 17 rather than in cell 01. Actually, we could have begun in cell 01, but it wouldn't have been good practice. Long experience has taught programmers that it is a good idea to leave some empty cells in front of a program. These provide a little "elbow room" if the earlier part of a program has to be backed up to insert a forgotten word. Maneuvering space should also be left at the end of a program for the same reason.

SECTION 9.

LOOPS

Consider the following program for counting. It will generate an output of 1-2-3-4-5 . . . and so on, up to any number you want.

Program No. 2: Counting

ADDRESS	CONTENTS
20	100
21	603
22	503
23	200
24	603
25	503
26	200
27	603
28	503
29	200
30	603
31	503
32	200
33	603
34	503

Like the program for addition, this program is not as interesting for *what* it does as for *how* it does it.

It won't be necessary to run this program through CARDIAC. We can spot a serious drawback simply by looking it over. One glance should be enough to show that it is much too long—fifteen words just to count up to five. A similar program for counting up to a million would fill the memory of even a large computer.

A more detailed examination is even more revealing. After the first word, the program repeats itself every three steps: 603, 503, 200; 603, 503, 200; and so on. Let's see what's happening:

ADDRESS	CONTENTS	COMMENTS
20	100	The contents of cell 00 (001) are put in the accumulator.
21	603	The accumulator contents (001) are copied into cell 03, (without being erased from the accumulator).
22	503	The contents of cell 03 are printed out. This is the first count.
23	200	The contents of cell 00 (001) are added to the contents of the accumulator (001) raising the sum to "002".
24	603	The accumulator's contents (002) are copied into cell 03, (without being erased from the accumulator).
25	503	The contents of cell 03 are printed out. This is the second count.
26	200	The contents of cell 00 (001) are added to the accumulator, raising sum to "003".
27	603	The accumulator's contents (003) are copied into cell 03, (without being erased from the accumulator).
28	503	The contents of cell 03 are printed out. This is the third count.
29	200	The contents of cell 00 are added to accumulator, raising sum to "004".
30	603	The accumulator's contents (004) are copied into cell 03.
31	503	The contents of cell 03 are printed out. This is the fourth count.
32	200	The contents of cell 00 are added to accumulator, raising sum to "005".
33	603	The accumulator's contents (005) are copied into cell 03.
34	503	The contents of cell 03 are printed out. This is the fifth count.

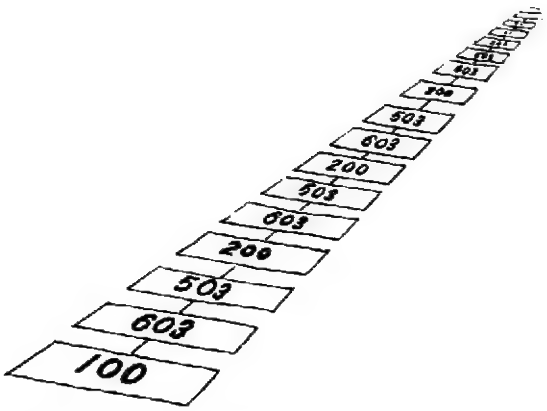


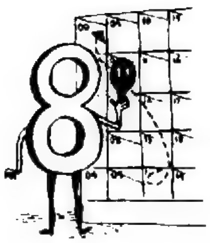
Fig. No. 13. Flow chart for counting program.

To make a computer count, as you saw, merely requires that it be programmed to keep adding one into the accumulator and to keep storing and printing out the sum.

Counting is, in fact, an important computer function and is often used in conjunction with other computer programs. The problem is: How do we make a computer repeat the counting cycle without having to spell our every step in the program? Obviously, we need another instruction—something that can be written into a program just once which will make the computer *loop* back to the beginning of each add-store-print cycle.

The Unconditional Jump

CARDIAC has just such an instruction. Operation Code number 8, the "jump" instruction, gets us out of this dilemma very nicely. It will send the bug back (or ahead) to any cell we want. In programming terminology, this is known as an *unconditional transfer*. It enables us to program a computer to loop through some repetitive sequence of operations without having to write each step of that sequence more than once.



But, reading about a loop makes it seem more complex than it really is, so let's actually run through one on CARDIAC.

Here is a "looped" program for counting. It can count indefinitely, yet it contains *only five words!* Run through it up to a count of three or four to prove to yourself that it works. Start with the bug in cell 21.

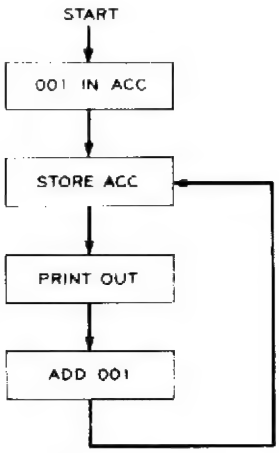


Fig. No. 14. Flow chart for counting program with a loop.

Program No. 3: Counting Program with Loop

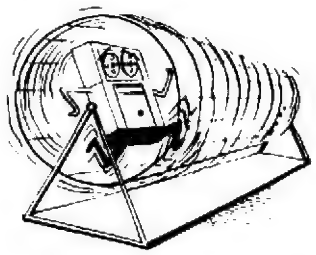
ADDRESS	CONTENTS	COMMENTS
21	100	Clear and add contents of cell 00.
22	603	Store contents of acc. in cell 03.
23	503	Print contents of cell 03.
24	200	Add contents of cell 00.
25	822	Jump to instruction in cell 22.

Operation code 8 embodies another handy feature (which wasn't used in the above program): In addition to letting the program counter jump out of sequence to begin a loop, it marks the counter's place so that it can return to where it left off when the computer is through looping. Operation code 8 does this by including a sub-instruction to record the bug's last address (before jumping) in cell 99. This procedure will be explained later in greater detail.

SECTION 10.

GETTING OUT OF LOOPS

The loop, as you can see, is a very useful part of a programmer's bag of tricks. So useful, that it's doubtful any programs are ever written without them. Even so, it may have occurred to you that, if programmers can put a computer *into* a loop, they must also have some means of getting it out again. A computer trapped in a loop is in serious trouble. Nor is this an uncommon programming error. What keeps it from being fatal is that programs are usually timed, and computers will automatically "dump" a program whose estimated running time has been exceeded.



In order to get out of a loop, a computer must be able to "make a decision" based on some predetermined criterion. This criterion can be determined by the programmer, but the decision-making ability must be built into the computer's hardware.

Since a looping computer keeps repeating the same instructions over and over again, the "decision" it makes must be to introduce a new instruction that will break the loop. Naturally, it can't be introduced at random. The computer must know exactly *when* to introduce the new instruction. It must recognize and respond to some predetermined change occurring within itself. The change can be (and often is) the change of a number's sign in the accumulator.

Say, for example, that a computer has been programmed to count backwards from 100. The program does this by adding 100 to the accumulator and then repeatedly subtracting "1" from it by means of a loop.

After 100 looped subtractions, the accumulator will pass through zero* to minus one. When this happens, the change of sign is de-

*Zero is arbitrarily defined as a positive number. Thus, if the count is to be a full 100, it must begin at 99.

tected by appropriate hardware and used to elicit a new instruction. This is precisely what happens in CARDIAC, and the new instruction is operation code number 3. Its mnemonic abbreviation is TAC—for "test accumulator contents." The program below puts this instruction to good use. In effect, it puts minus four in the accumulator and then keeps adding one to it until the accumulator reaches zero. Since zero is a positive number (for CARDIAC), the accumulator sign changes at this point, and the addition loop is broken. Run through this program, paying close attention to what happens when you change the accumulator sign. Also notice how op code no. 3 establishes a loop and then breaks it when the accumulator sign changes. After loading the program into the memory, put the bug in cell 20 to begin.

Program No. 4: Rocket-Launching Countdown

(Launch to occur when computer output generates 000)

ADDRESS	CONTENTS	COMMENTS
00	+001	Data.
19	−004	Data.
20	119	−004 to accumulator.
21	200	Add 001.
22	618	Store accum. in cell 18.
23	518	Print contents of cell 18.
24	321	Test acc. { If minus, jump to cell 21. If plus, go ahead to cell 25.
25	900	Halt and reset.

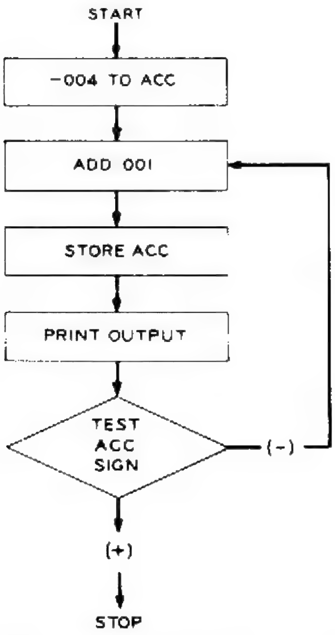
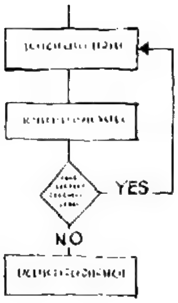


Fig. No. 15. Flow chart of rocket-launching countdown.

Conditional vs. Unconditional Transfers

As the previous two programs demonstrated, both the conditional transfer (op code 3) and the unconditional transfer (op code 8) modify the program counter's count. That is, they jump the bug to an out-of-sequence address. The major difference between the two instructions is that the unconditional transfer *always* elicts a jump, while the conditional transfer elicts a jump *only when the accumulator sign is negative*.



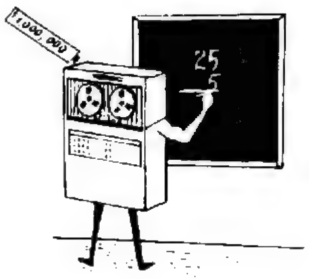
In addition to being a way of getting out of loops, the conditional transfer is a means of introducing alternate procedures depending on previously obtained results. Such alternate procedures are known as *branch points*. In flow charts, they are represented by a diamond shaped figure and always indicate the point at which some decision must be made. To see how they are used, go back to page 6 for another look at the "tire-changing" flow chart. When you are more experienced, you can try coding it into machine language for CARDIAC. Till then, you might give a moment's thought or two to the problem of converting the flow chart's "yes" or "no" answers into plus or minus signs for a computer's accumulator.

SECTION 11.

MULTIPLICATION

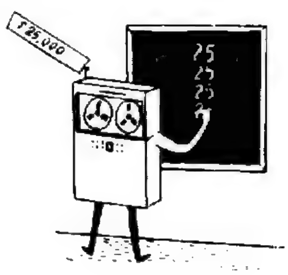
There are two ways of achieving multiplication in computers—expensively, or economically.

This is an oversimplification, of course; but it's true that the cost of a computer is usually a good indication of how it multiplies. As a rule, the larger, more expensive machines have built-in hardware that enables them to multiply (or divide) directly—pretty much the way we do.



Less expensive computers lack such hardware and have to resort to the more roundabout method of *repeated addition*. They store the larger of the two numbers to be multiplied in the accumulator and then repeatedly add that number to itself "n" times—"n" being equal to one less than the smaller number.

To multiply 25 by 5, for example; they put 25 into the accumulator and then, by means of a loop, add four more 25's to it.



Since CARDIAC is patterned after the more economical models, it, too, multiplies by repeated addition. This method is demonstrated by Program No. 5 on page 30.

Notice that this program uses op code 8 to generate the addition loop and op code 3 to get out of it. Notice also that "n" is tested during each loop, and that the loop is not broken until "n" turns negative. This method is known as a ***loop with an index***, where the index is equal to "n". The same method can be used in *any* program calling for some particular procedure to be repeated "n" times. Since "n" is fed in as *data*, rather than as an integral part of the program, such programs do not have to be revised when "n" is changed.

You can use any 2-digit number for the multiplicand, but it's best to use a smaller number for the multiplier, since this will determine how often you must repeat the loop.

Before beginning, copy the program into the indicated memory cells, and place the bug in cell 07. Then write the multiplicand on input card number 1 and the multiplier on card number 2. What would happen if you reversed their order?

Program No. 5: Multiplication by a Single-Digit Multiplier

(multiplier) A x BC (multiplicand)

ADDRESS	CONTENTS	COMMENT	
07	068	Read "BC" into cell 68.	
08	404	Clear* acc.	Clearing cell 69 for future storage of sum.
09	669	Store acc. (zero) in cell 69.	
10	070	Read "A" into cell 70. This will be "n".	
11	170	"n" to acc.	
12	700	Subtract 1 from "n"	
13	670	Store revised "n."	
14	319	Test acc. sign.	
15	169	Clear acc. Enter contents of cell 69 (previous sum).	
16	268	Add "BC" to acc.	
17	669	Store revised sum in cell 69.	
18	811	Jump back to cell 11.	
19	569	Print (product of "A" x "BC").	
20	900	Halt and reset.	

*See Section 12 for explanation of how this instruction clears the accumulator.

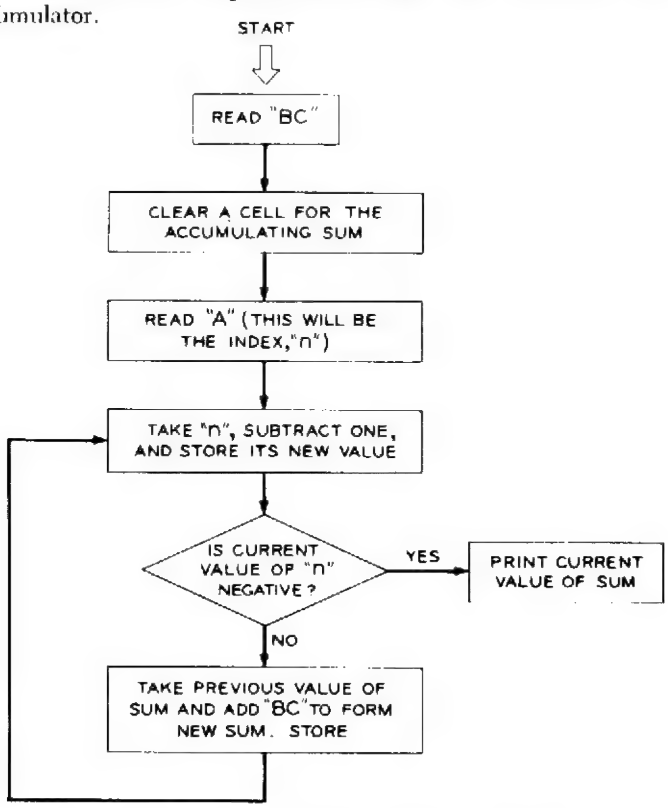

START
READ "BC"
CLEAR A CELL FOR THE ACCUMULATING SUM
READ "A" (THIS WILL BE THE INDEX,"n")
TAKE "n", SUBTRACT ONE, AND STORE ITS NEW VALUE
IS CURRENT VALUE OF "n" NEGATIVE ?
YES
PRINT CURRENT VALUE OF SUM
NO
TAKE PREVIOUS VALUE OF SUM AND ADD "BC" TO FORM NEW SUM. STORE

Fig. No. 16. Flow chart of single-digit multiplication.

SECTION 12.

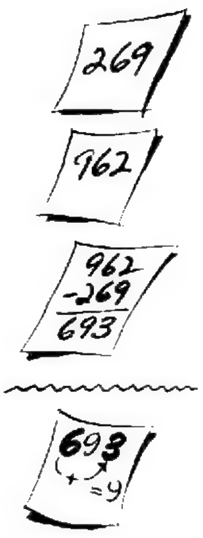
SHIFTING DIGITS

The Magic "Nines" Trick

Here's a trick you can bedazzle a friend with—provided he hasn't read this far yet.

Ask your friend to write any 3-digit number in which no two digits are repeated. Don't let him show it to you. Then, tell him to write them in reverse order. Now, have him subtract the smaller of the two 3-digit numbers from the larger. Ask him for the last digit of the difference. When he gives it to you, you immediately tell him the whole number. How? Read on.

It's all very simple. The middle digit will *always* be a nine, and the two end digits will *always* add up to nine. Thus, all you have to do is subtract the given digit from nine to get the first digit. Naturally, the trick is more impressive when done quickly and with a little showmanship.



What has all this to do with CARDIAC and computers? Well..., not much. It's just a sneaky way of introducing the last of CARDIAC's ten instructions.

Operation Code 4—The Shift Instruction

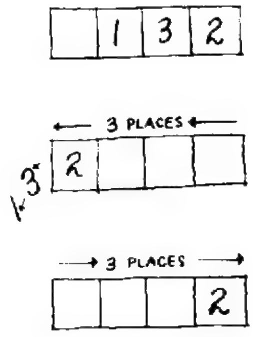
If you want to use CARDIAC for the "nines" trick instead of a friend, it will have to be capable of reversing the order of a given number. As the next two programs will demonstrate, CARDIAC can not only reverse numbers, it can manipulate them in various other ways, as well. To do so, it uses op code 4—the "shift" instruction.

We've saved this instruction for last, not because it is the least important, but because it is probably most alien to your everyday experience.

What it does, in brief, is shift a number in the accumulator to the left "x" number of places and then to the right "y" number of places. The value of "x" and "y" is specified by the second and third digits of the shift instruction. *This is the only instruction whose last two digits do not refer to an address in the memory.*

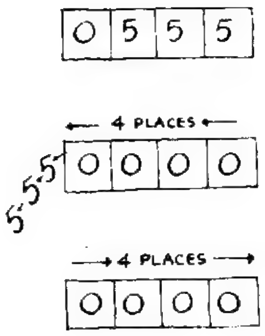
Before you can use the shift instruction correctly, you must understand two things:

(1) Digits overflowing the accumulator are irretrievable. Let's say you have 132 in the accumulator and the instruction register reads "433." This calls for the number in the accumulator to be shifted left three places and then right, three places. Do you finish up exactly where you began? Not at all! When the lefthand shift pushes the 1 and the 3 out of the accumulator, they are gone for good. During the righthand shift, only the 2 is returned to its former location.



(2) There are no such things as "blank" spaces in the accumulator (or in any other computer register). When a digit is moved out, it is immediately replaced by a zero. If, for example, the accumulator holds 555, and a four-place shift to the left and right is called for, the resulting contents will be 0000.

The following program will give you ample practice in the use of the shift instruction. Write it into the indicated memory cells and then write the 3-digit number to be reversed on input card number one. Start with the bug in cell 15.



Program No. 6: Reversing the Order of a Number "abc"

ADDRESS	CONTENTS	COMMENTS
15	039	Read "abc" into cell 39.
16	139	CLA "abc".
17	431	Shift acc. to produce "c00".
18	640	Store acc. in cell 40.
19	139	CLA "abc".
20	413	Shift acc. to produce "00a".
21	240	Add contents cell 40 to produce "c0a" in acc.
22	640	Store acc. in cell 40.
23	139	CLA "abc".
24	423	Shift acc. to produce "00b".
25	410	Shift acc. to produce "0b0".
26	240	Add contents cell 40 to produce "cba" in acc.
27	640	Store acc. in cell 40.
28	540	Print contents cell 40.
29	900	Halt and reset.

SECTION 13.

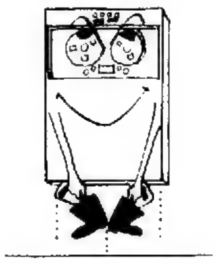
BOOTSTRAPS AND LOADING PROGRAMS

Up to now, we've had no trouble loading programs into CARDIAC. We've simply copied them into the indicated memory cells and gone merrily on our way. Unfortunately, this is not the way it's done with real computers. Programs have to be loaded through the input, just like data. What's more, each word has to be steered to the proper address. This calls for a special *loading program*.

New, or repaired, computers face an additional problem. When a new machine is first turned on, all of its registers contain garbled nonsense. As we mentioned earlier, there can be no "blanks" in any computer register. The flip-flop circuits, memory cores, and other hardware used to store binary one's and zero's must *always* indicate one or the other. *Which* way they flip when a new machine is plugged in is purely arbitrary, so the resulting word combinations are meaningless.

Before a program can be loaded, the contents of some of the registers must be organized. Like an infant, the new computer must learn at least a few words before it can begin to talk coherently. This calls for special hardware.

Some machines have reset buttons for setting the program counter to zero and for clearing the accumulator. Others have a long row of input buttons for manually inserting words, bit by bit. But, whatever the method, new computers must, in a very real sense, lift themselves up by their own bootstraps to get going. Not surprisingly, the process is called *bootstrapping*.



In the early days of computers, there were informal competitions to see who could cut bootstrapping operations to a minimum. CARDIAC would have won any such contest hands down. The only special "hardware" it uses is the word "001" permanently stored in cell 00. And its bootstrapping routine consists of only two words: "002" and "800." We'll see how these manipulate a loading program in just a moment.

Loading Programs

A loading program is simply the program we wish to get into the memory, interlaced with suitable input instructions and addresses. In a real computer, it usually takes other, more efficient, forms. But, for CARDIAC, interlacing loading instructions and program words is perfectly adequate. The only problem is, "How do you ever get ahead if you have to repeatedly write a set of instructions for every program word you want to store? "Well, you use a loop, and that's where the bootstrap routine and the "001" permanently wired into CARDIAC's memory come in.

Using our first addition program as an example, here's how the whole thing looks:

Program No. 7: Bootstrap and Loading Program for Addition

INPUT CARD	CONTENTS	COMMENTS
	001	Bootstrap (already in memory cell 00).
1	002	"
2	800	"
3	010	Addressing instruction.
4	017	Program word.
5	011	Addressing instruction.
6	018	Program word.
7	012	Addressing instruction.
8	117	Program word.
9	013	Addressing instruction.
10	218	Program word.
11	014	Addressing instruction.
12	619	Program word.
13	015	Addressing instruction.
14	519	Program word.
15	016	Addressing instruction.
16	900	Program word.
17	BLANK	Signals end of program.
18	___	Data word (number to be added).
19	___	" " " " " "

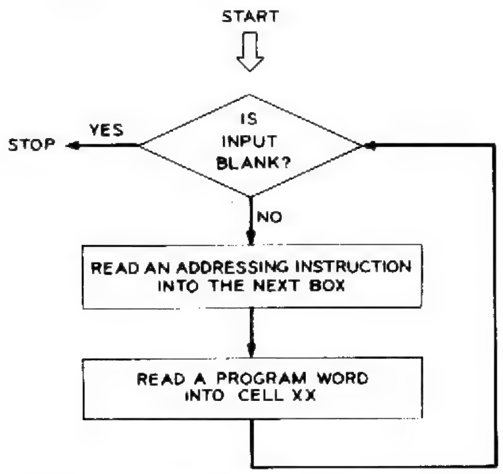

START
IS INPUT BLANK?
YES
STOP
NO
READ AN ADDRESSING INSTRUCTION INTO THE NEXT BOX
READ A PROGRAM WORD INTO CELL XX

Fig. No. 17. Flow chart for Program No. 7.

Beginning with the bootstrap routine, write the program on input cards, insert the bug in cell 00, and begin loading. It probably won't be necessary to load the entire program to see how it works. Nor will it be necessary to load future programs this way. As long as you understand that you're taking a short cut, you can keep writing them directly into the memory.

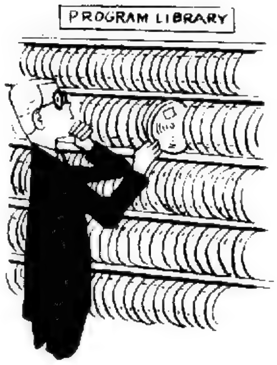
SECTION 14.

SUBROUTINES

In the early days of computers, programmers had no libraries of taped programs to turn to. Every time they tackled a new program, they started from scratch.

If part of a program called for sines, cosines, cube roots, or any of a thousand other common mathematical routines, they had to write every step of that routine.

Before long, they realized they were tediously recreating the same routines over and over again. It was an enormous waste of time and creative energy; and, with that realization, was born the concept of the subroutine.



What They Are

A subroutine is simply a piece of a program—usually stored on magnetic tape in such a fashion that it can be easily used by any program.

There are subroutines available covering everything from commonplace sines and cosines to functions so esoteric as the *Kramers-Kronig Analysis of Reflectance.* Their beauty lies in the fact that, once written and recorded, they need never be written again.

Sometimes, a full program proves to be useful in the partial solution of a larger problem. In such cases, it can be used as a subroutine for the larger program.

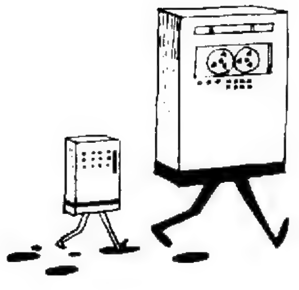
Because subroutines proliferate so quickly, catalogs listing them are constantly being updated and reissued. For obvious reasons, these are studied by programmers with the same kind of zealous attention horse players devote to racing forms—and, usually, with far more profit.

Calling Sequences

Programs involving subroutines are not written with big gaps for the subroutines to be tucked into. Instead, they make use of *calling sequences*—instructions that call for the desired subroutine and then jump the program counter to the subroutine's entrance address. The calling sequence also has provisions for transferring data to and from the subroutine.

Another requirement of the calling sequence is that it note the address to which the main program will return after the subroutine has run its course. CARDIAC uses op code 8 to do this. In the next program, you will make full use of this instruction for the first time —including the part that asks you to "Write bug's cell No. in cell 99."

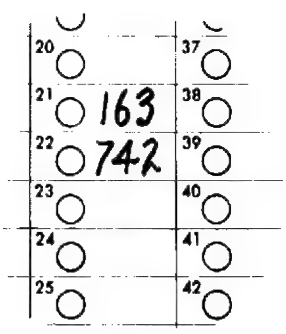
Double Precision



The precision of a numerical description (of anything) is a function of the number of digits used. If you say that something is 3.62958 inches long, you are being twice as precise as the fellow who limits himself to saying it is 3.63 inches.

Thus, the precision of computers would seem to be limited to the maximum word length their hardware can handle. Fortunately, this isn't entirely true. Curiously enough, small computers can be programmed to *simulate* the greater capability of their big brothers. But they do so at the expense of speed, because they must perform many more operations to accomplish the same things. Naturally, there are practical limits to how far such simulation can be pushed. In terms of cost per-operation, it is still generally cheaper to solve big problems on big computers.

Double-Precision Subroutine for CARDIAC



The following program for double-precision addition will illustrate all of the points made above. It enables you to run 6-digit arithmetic through CARDIAC's 3-digit hardware.*

Basically, the method used to handle 6-digit numbers is to store the three *most* significant digits in one location, and the three *least* significant digits in another. The two locations will be adjacent, with the most significant digits going into an *odd*-numbered cell and the least significant digits going into the following *even*-numbered cell. For example 163,742 can be stored by putting 163 in cell 21, and 742 in cell 22. The details of all this bookkeeping will be handled by the subroutine.

Incidentally, we could write 6-digit subroutines for each of CARDIAC's ten instructions. If we did, any single-precision program could then be converted to double-precision by substituting double-precision subroutines for each ordinary instruction.

*When you get through, compare the time it took to add two 6-digit numbers with the time it took to add two 3-digit numbers. Only then will you appreciate the true significance of ". . . at the expense of speed."

Copy the subroutine and main program into the indicated locations. Then write the two numbers to be added onto input cards 1 through 4. Write the three most significant digits of numbers A and B on cards 1 and 3, and the least significant digits on cards 2 and 4. Start with the bug in cell 50.

Program No. 8: Subroutine for "A" + "B" = SUM

ADDRESS	CONTENTS	COMMENT
86	199	Prepare exit.
87	694	
88	196	Add least significant digits.
89	298	
90	698	
91	403	Shift overflow right and add most significant digits.
92	295	
93	297	
94	8 _ _	Return to program. (_ _ will be the address of the last instruction plus one.)

MAIN PROGRAM

ADDRESS	CONTENTS	COMMENT
50	095	Input and calling sequence.
51	096	
52	097	
53	098	
54	886	
55	659	Output.
56	559	
57	598	
58	900	Halt and reset.

SECTION 15.

DEVELOPING PROGRAMS

Although we have run several programs through CARDIAC,* so far, we've devoted very little attention to the difficult business of preparing a program. In this chapter, we will briefly examine some of the steps involved in going from a problem to a program.

First of all, it is necessary to determine *exactly what* the problem is. This can be done only by stating the problem in precise, unambiguous terms. Occasionally, a problem never gets past this step, because closer examination proves it to be too inherently vague for computer solution.

The next step is to analyze the problem with the aim of finding a method of solution, or *algorithm*.

The dictionary defines algorithm as "a rule of procedure for solving a recurrent mathematical problem . . . ," but computer people use the word in a more general sense. They think of an algorithm as a precisely stated set of rules for accomplishing *any* task. By precisely stated, they mean step-by-step rules that can be followed without intuition or overall understanding–like CARDIAC's op codes.

The statement must also take into account the nature of the device that will respond to the rules. An algorithm adequate for humans is rarely precise enough for a computer. A flow chart is an example of an algorithm that can be followed by a human; a *program* is an algorithm for a computer.

Flow charts are written in plain language. Programs are written in machine language (as are CARDIAC's), or in any one of the many higher-level computer languages now available.

Hence, language is the third basic ingredient of program development. In order to talk about a problem, we must use some kind of language. If a computer is to solve the problem, that language must be one the computer can "understand." The machine lan-

*If you have executed all the previous programs, it may not be necessary to run any more through CARDIAC. By now, it should be sufficient to write a program in CARDIAC's memory and run through it mentally. A list of op codes has been printed next to the memory to facilitate this kind of procedure.

guage used by CARDIAC looks a little strange to us (so strange, we need an *instruction-decoder*) but it is perfectly clear to CARDIAC.

To sum up, the three major steps in developing a program are (1) precisely defining the problem, (2) finding the right algorithm, and (3) expressing the algorithm in the correct language for the computer that will execute the program.

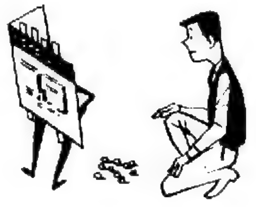
Developing a Program to Play Single-Pile Nim

Much of the foregoing can be illustrated by our development of a program to play Single-pile Nim.

We've chosen this game, because, like most games, it can be clearly defined. Its rules, strategies and moves can all be precisely spelled out. In addition, it's not so large a problem that the whole of it can't be seen fairly easily. Finally, we chose single-pile Nim, because it is a game in which numbers occur naturally. This makes it relatively easy to invent a simple language for CARDIAC.

We will analyze the game and then write a series of programs that permit CARDIAC to play as our opponent. Each new program will enable CARDIAC to perform a different job called for by the game. Some of these jobs are not covered by CARDIAC's ten instructions. This means we'll have to use some ingenuity in devising the programs.

Rules of Single-Pile Nim



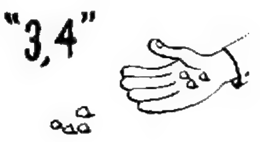
Ten pebbles are placed in a single pile between the two players. During a player's turn, he may remove one, two, or three pebbles —provided that his opponent has not removed a similar number of pebbles during *his* turn. In other words, if your opponent (CARDIAC) has just taken two pebbles, you may take one or three, but not two.

A player loses when he cannot move because:

(1) There are no pebbles left for him to remove.

(2) There is only one pebble left, and his opponent has just taken one pebble.

The Strategy



Any move can be defined by a two-digit number, such as 3,4. The first digit represents the number of pebbles taken during the move, and the second digit represents the number of pebbles left in the pile. For example: If the first player to move takes one pebble (from the original ten), the move is defined as 1,9. If the second player now takes two pebbles (leaving seven), his move is defined as 2,7.

The game ends with any of four final winning moves: 1,0; 2,0; 3,0; and 1,1.

Analysis

To eliminate ambiguity, one seemingly obvious point must be stressed: A move must be either a winning move or a losing move. If a move leaves either player with a choice of *only* losing responses, this is clear enough. But, if a move leaves either player with a choice of losing *or* winning responses, we will assume (for purposes of this analysis) that the correct, winning choice will always be made.

From this assumption and our knowledge of the previously stated rules, we can now work backwards from the four *final* winning moves to identify every possible move as a winner or a loser.

Thus, just as 1,0 is a winning move, because it leaves the opponent without a permissible move, 2,1 and 3,1 are losing moves because they permit the next player to make the winning 1,0 move.

Similarly, we can work backwards from winning move 2,0 to derive the losing moves 1,2 and 3,2. We can also work backwards from winning move 3,0 to losing moves, 1,3 and 2,3. And, finally, we can work back from winner 1,1 to losers 2,2 and 3,2.

Since it will be necessary to define *all* possible moves, it's best we begin tabulating our results:

WINNING MOVES			LOSING MOVES		
1,0	2,0	3,0			
1,1				2,1	3,1
			1,2	2,2	3,2
			1,3	2,3	

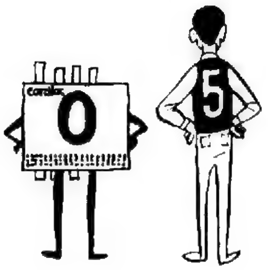
Still working backwards (this time, from losing moves to winning moves), we can derive five more winners: 3,3; 1,4; 2,4; 3,4; and 1,5. We can continue working back from these five new winners to derive seven more losers: 2,5; 3,5; 1,6; 2,6; 3,6; 1,7; and 2,7. And, finally, working back from these losers, we come up with five more winners to complete our table as follows:

WINNING MOVES			LOSING MOVES		
1,0	2,0	3,0			
1,1				2,1	3,1
			1,2	2,2	3,2
		3,3	1,3	2,3	
1,4	2,4	3,4			
1,5				2,5	3,5
			1,6	2,6	3,6
		3,7	1,7	2,7	
1,8	2,8				
1,9					

Close examination of our table shows that the first player to go should win, since all three possible opening moves (1,9; 2,8; 3,7) are in the winning column.

Because we now know the value of every possible move, we can store this information in CARDIAC's memory and devise a simple program to look up the information as needed. This is known as a *table look-up program.*

CARDIAC, as we know, responds to three digit instructions. We use the first digit of each instruction to identify the player. Zero will represent CARDIAC, and 5 will represent CARDIAC's human opponent.



The second digit of each instruction will specify how many pebbles are taken, and the third digit, the number of pebbles left in the pile. As examples, here are five moves of a typical game:

028	CARDIAC takes two pebbles, leaving eight in pile.
517	Player takes one pebble, leaving seven.
034	CARDIAC takes three, leaves four.
513	Player takes 1, leaves three.
030	CARDIAC takes three, leaves none—wins.

Using these conventions and table look-up allows us to devise what may be the shortest game program ever written. Here it is in its entirety:

Program No. 9: Single-Pile Nim (Ten Pebbles)

ADDRESS	CONTENTS	COMMENT
00	001	Read an input card (this instruction is wired in).
01	529	This is used only when CARDIAC plays first. Cell 01 must be reset to 529 for each new game.
02	900	Halt and reset.

The following look-up table must be entered in CARDIAC's memory.

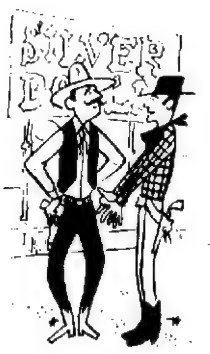
ADDRESS	CONTENTS	ADDRESS	CONTENTS
10	000	24	013
11	001	25	014
12	020	26	015
13	030	27	034
14	022	28	017
15	023	29	019
16	033	30	000
17	034	31	010
18	026	32	020
19	027	33	012
20	000	34	013
21	010	35	014
22	011	36	024
23	030	37	025

To play the game, write each of your moves on an input card. CARDIAC will respond by printing its moves out on an output card.

If you are to go first, begin with the bug in cell 00. If CARDIAC is to make the first move, begin with the bug in cell 01. When CARDIAC goes first, it is unbeatable. You'll probably win if you go first, but a bad play could cost you the game.

Improving the Game

Knowing that the first player to move was unbeatable must have taken some of the joy out of the game. You may even have felt a little like the gambler who was hailed by a friend on his way to the local casino.



"Where you headed?" asked the friend.

"Oh, I figure to try my luck at the Silver Dollar Casino."

"You danged fool, don't you know the game is crooked"

"Sure I do," said the gambler, without missing a stride, "but what can I do? It's the only game in town!"

Well, now we'd like to make it possible to beat the first player—though not by lowering the quality of CARDIAC's play. CARDIAC should still be programmed to always make the best move possible. And, to increase the element of chance, it should also be programmed to give its opponent the greatest opportunity to make a bad move.

In addition, we'd like to make CARDIAC work a little harder—perhaps by doing the arithmetic necessary to determine how many pebbles are left in the pile after each move.

How can we do all this? Well, for a start, we can extend our table of winners and losers beyond the previous limit of ten.

Extended Table of Moves

WINNING			LOSING		
1,0	2,0	3,0			
1,1				2,1	3,1
			1,2	2,2	3,2
		3,3	1,3	2,3	
1,4	2,4	3,4			
1,5				2,5	3,5
			1,6	2,6	3,6
		3,7	1,7	2,7	
1,8	2,8	3,8			
1,9				2,9	3,9
			1,10	2,10	3,10
		3,11	1,11	2,11	
1,12	2,12	3,12			
1,13					

Our new table reveals two interesting things: First of all, we can see that raising the initial number of pebbles makes it possible to make an opening move that is a loser. With 13 pebbles, for instance, an unwary first player has an opportunity to make either of two wrong moves: 2,11 or 3,10. Thus, raising the initial number of pebbles will accomplish one of our purposes.

The second thing we notice about our new table is its periodicity, which shows up at first as a geometric pattern:

```
XXX  XXX  XXX
XXX                             XXX  XXX
                           XXX  XXX  XXX
          XXX              XXX  XXX
XXX  XXX  XXX
XXX                             XXX  XXX
                           XXX  XXX  XXX
          XXX              XXX  XXX
XXX  XXX  XXX
XXX                             XXX  XXX
                           XXX  XXX  XXX
          XXX              XXX  XXX
XXX  XXX  XXX
XXX
```

Closer examination reveals that the table is periodic in fours. That is, adding or subtracting multiples of four to the pile doesn't affect the results of any play. For example, the moves, 2,1; 2,5; 2,9; 2,13; etc. are all losers. The proper reply to each of them is to take one pebble, making for the winning moves 1,0; 1,4; 1,8, and 1,12.

We can take advantage of this periodicity to program CARDIAC to play with any number of pebbles up to 999. What we'll do is have CARDIAC play *as though* it could subtract a sufficient number of four's from the pile until the pile is reduced to a number smaller than four. CARDIAC will then select its move *as though* the pile contained only this number of pebbles. In other words, regardless of how large the pile really is, CARDIAC will always move as though it contains three pebbles or less. This procedure will eliminate the need for storing 999 replies in CARDIAC's memory. We will, in fact, need only sixteen entries in CARDIAC's look-up table of plays.

Strategy

In developing our strategy, we must take the following three considerations into account:

1. *Determining CARDIAC's reply when its opponent makes a losing move that leaves an odd number of pebbles.*

Our table shows us that, of the two possible replies to losing moves that leave an *odd* number of pebbles in the pile, one is always a winner, and the other is always a loser. In all such cases, CARDIAC's choice of reply will, of course, always be the winner. For example, the two possible replies to the losing move 2,11 are 3,8 (a winner) and 1,10 (a loser). CARDIAC's reply will be 3,8.

2. *Determining CARDIAC's reply when its opponent makes a losing move that leaves an even number of pebbles.* Further examination of our table shows us that, when there are two possible replies to losing moves that leave an *even* number of pebbles, both are always winners. The only instances in which two replies *aren't* possible are those contained in the first group of losing moves. Two replies aren't always possible here for the simple reason that most of the moves don't leave enough pebbles. A move like 2,1, for example, permits only one reply—1,0.

This first group, then, is a special case. To keep things simple, and to reduce the strain on CARDIAC's memory, we will determine its replies on the basis of this special group. Because of the table's periodicity, we can then use these same choices for all subsequent groups. For example, the only possible reply to the losing move 1,2 is 2,0. Hence, CARDIAC's reply to all periodically equivalent moves such as 1,6; 1,10; 1,14 etc. will be to take two pebbles.

3. *Providing CARDIAC's opponent with the maximum opportunity for making a bad move.* Suppose CARDIAC's opponent makes a winning move. CARDIAC's choice of reply should give him the greatest opportunity to make a *losing* move the next time. If the player's move is 1,8, for example, we can reply with either 2,6 or 3,5. If we choose 2,6, the player's subsequent move *must* be a winner, regardless of whether he chooses to reply, 1,5 or 3,3. On the other hand, if our reply is 3,5, the player has a chance to make a losing move, (2,3), as well as a winning move, (1,4). Hence, 3,5 is our best move. We will choose our replies to the player's winning moves in the same way, wherever possible.

Table of Moves with CARDIAC's Replies

With our strategy established, we can decide CARDIAC's reply to every move. Parenthetically adding these replies to our tables of moves will make it that much more complete and informative.

WINNING			LOSING		
1,0(*)	2,0(*)	3,0(*)			
1,1(*)				2,1(1)	3,1(1)
			1,2(2)	2,2(1)	3,2(2)
		3,3(2)	1,3(3)	2,3(3)	
1,4(3)	2,4(1)	3,4(1)			
1,5(2)				2,5(1)	3,5(1)
			1,6(2)	2,6(1)	3,6(2)
		3,7(2)	1,7(3)	2,7(3)	
1,8(3)	2,8(1)	3,8(1)			
1,9(2)				2,9(1)	3,9(1)
			1,10(2)	2,10(1)	3,10(2)
		3,11(2)	1,11(3)	2,11(3)	
1,12(3)	2,12(1)	3,12(1)			
1,13(2)					

*Game over.

Developing the Flow Chart

Now that we've determined our strategy and CARDIAC's replies, we're ready to work up a flow chart. We can do this in two stages —the first showing what CARDIAC is to do in broad outline; and the second filling in some of the details of how it is to do it.

Broadly, some of the things we want CARDIAC to do are

1. Read the number of pebbles taken by the player. We will call this number "P".
2. Keep track of the pile's residue—that is, the number of pebbles that would be left in the pile if the highest possible multiple of four were subtracted from it. We will call this residue "R".
3. Keep track of the actual number of pebbles in the pile and print that number after every move. We will call this number "N".
4. Print the number of pebbles CARDIAC takes. We will call this number "C".

We can now draw up our first broad, or *macro* flow chart:

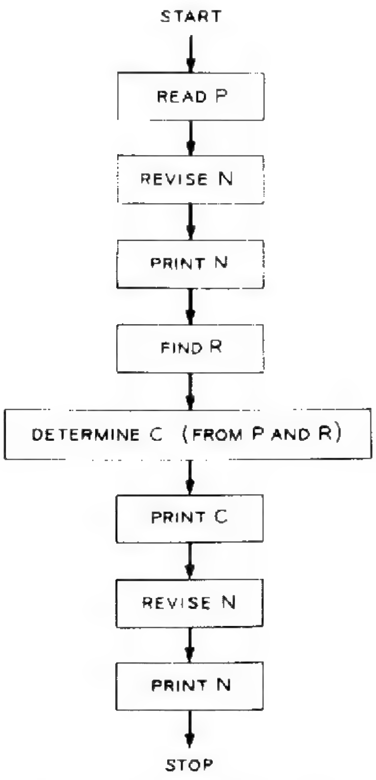


Fig. No. 18. Macro flow chart
for improved Nim game.

Converting most of the steps of our flow chart into program instructions will be a pretty straightforward procedure. "Read P," for example, will obviously dictate the use of an input instruction followed by the address of whatever cell P is to be read into. Similarly, the first "Revise N" step simply calls for a CLA instruction to put N into the accumulator; then, a SUB instruction to subtract P from that N; and, finally, a STO instruction to store the revised N in some specific address.

However, the method of implementing the two steps, "Find R" and "Determine C" is neither simple nor obvious. Hence, a detailed or *micro,* flow chart covering both these steps will be necessary.

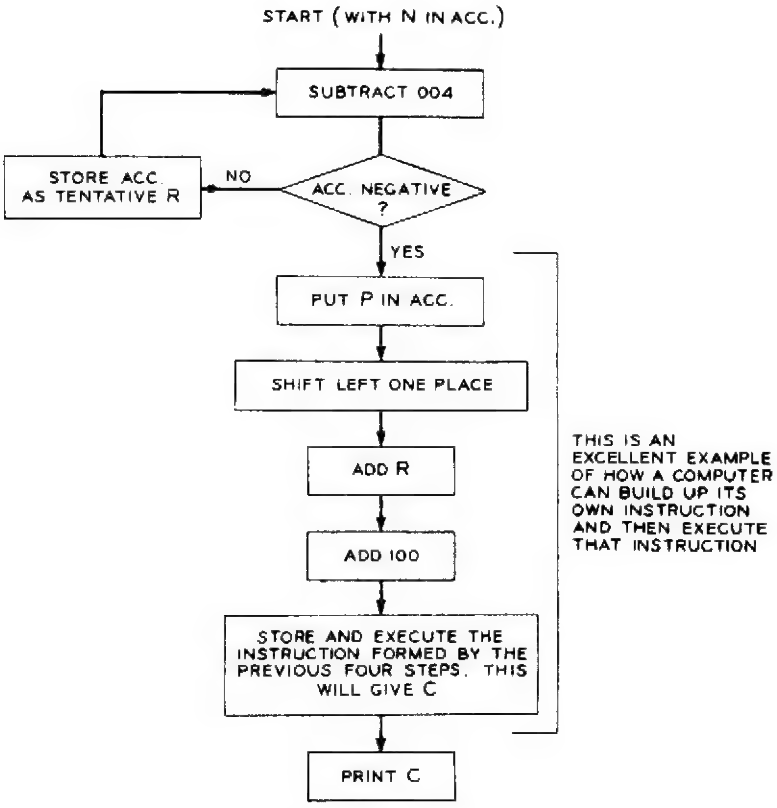

START (WITH N IN ACC.)
SUBTRACT 004
ACC. NEGATIVE ?
NO
STORE ACC. AS TENTATIVE R
YES
PUT P IN ACC.
SHIFT LEFT ONE PLACE
ADD R
ADD 100
STORE AND EXECUTE THE INSTRUCTION FORMED BY THE PREVIOUS FOUR STEPS. THIS WILL GIVE C
PRINT C
THIS IS AN EXCELLENT EXAMPLE OF HOW A COMPUTER CAN BUILD UP ITS OWN INSTRUCTION AND THEN EXECUTE THAT INSTRUCTION

Fig. No. 19. Micro flow chart finding R and C.

Program and Directions

With our flow charts thus worked out, it is now only a matter of mechanically *coding* them into program instructions for CARDIAC and then choosing suitable locations in which to store them. Similarly, since our strategy has enabled us to determine CARDIAC's best replies, we have only to choose some other locations in which to store these.

NOTE: If all of the foregoing has seemed rather lengthy and involved for what may appear a relatively short progam, we can only say that that's the way it is with programming. Almost all of the work and creative agony is expended in analyzing the problem, determining a method of solution, and resolving it into computer digestible fragments.

Program No. 10: Improved Nim Game

(If player is to move first, start with bug in cell 52. If CARDIAC moves first, start with cell 53)

ADDRESS	CONTENTS	COMMENTS	
52	015	Read P	
53	114		
54	715	Revise N	
55	614		
56	514	Print N	
57	718	Subtract 4	
58	361	Acc. negative?	FIND R
59	617	Store	
60	857	Jump	
61	115	Load P (00P in acc.)	
62	410	Shift left (0P0 in acc.)	
63	217	Add R (0PR in acc.)	
64	219	Add 100 (1PR in acc.)	
65	666	Store (1PR in Cell 66)	
66	100	Load C (CLA contents of Cell PR)	
67	616	Store C	
68	516	Print C	
69	114		
70	716	Revise N	
71	614		
72	514	Print N	
73	952	Halt (Return to 52)	

All moves will be indicated by a three-digit number 001 to 003 stating the quantity of pebbles removed from the pile. The player will write his moves on the input cards. CARDIAC will print its moves on output cards in the same way. Its moves should now be entered into the memory locations shown below.

Table of CARDIAC's Moves

ADDRESS	CONTENTS
00	001
01	001
02	002
03	003
10	003
11	002
12	002
13	003
20	001
21	001
22	001
23	003
30	001
31	001
32	002
33	002

Table of Constants and Variables

ADDRESS	CONTENTS	DESCRIPTION
14	N(start)	N = number of pebbles in pile
15	000	P = number of pebbles player takes
16	000	C = number of pebbles CARDIAC takes
17	N(start)	R = residue
18	004	The quantity 4 to be subtracted
19	100	Op code 1, CLA

Directions

(1) Load the program and the table of CARDIAC's moves into the memory.

(2) Load the table of constants and variables into the memory. Note that the starting quantity, N, is loaded into the residue cell 17 as well as cell 14. This enables the program to work even for starting quantities of N which are less than four.

(3) Start with the bug in cell 53, if CARDIAC is to go first. If the player is to go first, start with the bug in cell 52. The player is to write the number of pebbles he takes (001, 002, or 003) on input cards.

SECTION 16.

ASSEMBLERS AND COMPILERS...PROGRAMS FOR WRITING PROGRAMS

It takes only a glance at CARDIAC's list of op codes to see that they are *arithmetic* oriented. Even so, we managed to program it to do something as non-arithmetical as play a game. With enough ingenuity, we could have programmed it to do almost any of the much more complex tasks being done by real computers.

However, writing such programs for CARDIAC—let alone *executing* them—would be exceedingly tedious. As shown by our program for improved Nim, one of the major problems is to keep track of where things are stored. We have to remember not only *what* N and P and C and R represent, but also *where* they are stored. This kind of internal bookkeeping can be very difficult to keep straight—particularly with involved programs.

In our last program, for example, revising N required three instructions that caused (1) N to be put into the accumulator, (2) P to be subtracted from N, and (3) the result to be stored. To write those instructions, it was necessary to remember what addresses had been assigned to N and P, as well as the necessary op codes. Once we got all that straight, we wrote the instructions 114, 715, 614.

Now, if computers could somehow be persuaded to do such bookkeeping for us, programming would be much simpler. In place of 114, for example, we could simply write CLA N, and let the computer worry about where N was located and what the op code for CLA was.

Fortunately, computers can be programmed to do just that. The programs used to do so are called *assemblers*. Actually, there is considerably more to what assemblers do than we've been able to go into here. Among other things, they also attend to subroutines and calling sequences. All in all, they are a tremendous aid to programmers—so much so, that they are usually left in a computer's memory permanently.

Compilers

Assemblers, then, are programs *for writing programs*. They take a program written in assembly language and rewrite it in the more fundamental alphabet of machine language. Thus, assembly language is on a higher level than machine language in the sense that its meaning is more easily understood by humans. Happily, the process of writing a program to translate a higher into a lower level language can be carried one step further with *compilers*.

Compilers are used to rewrite programs of a still higher level than assemblers into assembly language. They go considerably further than assemblers towards eliminating programming drudgery. Whereas assembler programs necessitate writing a separate instruction for each machine-language instruction to be produced, compilers can translate one compiler-language instruction into *several* assembly-language instructions. Thus, the number of statements a programmer must write is vastly reduced.

One of the best known and most widely used compilers is FORTRAN—short for FORmula TRANslator. It is mathematically oriented, and its language resembles that of algebra.

With FORTRAN, a programmer wishing to solve an equation such as $J=K+M-N+2$ would write the equation exactly that way: $J=K+M-N+2$. The FORTRAN compiler would then translate that statement into suitable assembly-language instructions. For CARDIAC, these instructions would be:

```
CLA  K
ADD  M
SUB  N
ADD  2
STO  J
```

The assembler would next translate these into machine-language instructions and assign them locations in the memory like this:

ADDRESS	CONTENTS
20	151
21	252
22	753
23	270
24	650

At this point, the computer would cause a set of punched cards containing the complete program to be produced. If requested to do so, it would simultaneously print out the program in all three languages so the programmer could check it. The printout would also include a table of temporary storage assignments and a table of literals as follows:

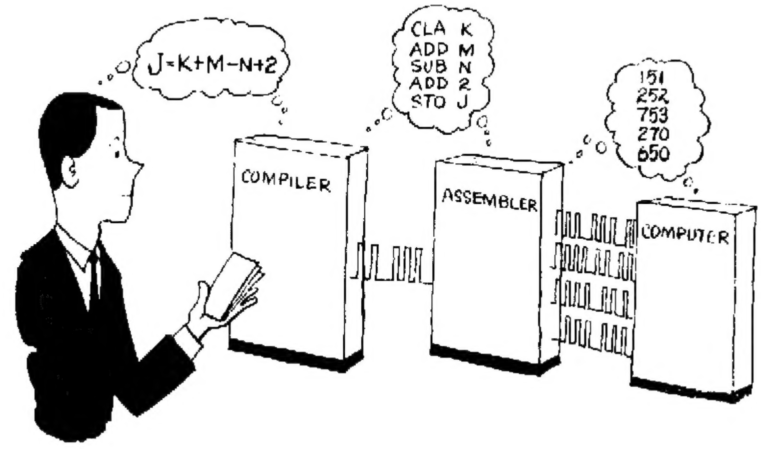


Table of Temporary Storage Assignments

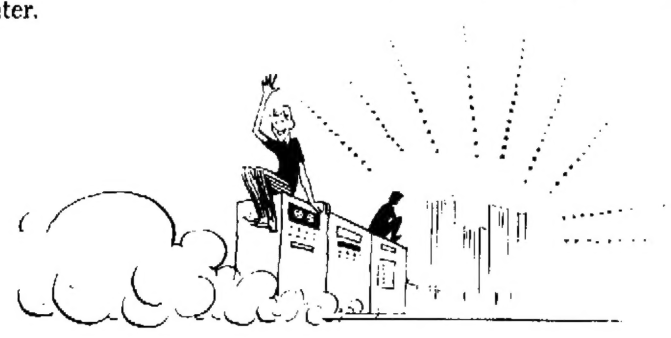
ADDRESS	CONTENTS
50	J
51	K
52	M
53	N

TABLE OF LITERALS

70	002

If any routine errors were made, *diagnostics* might also be included in the printout. These are comments on commonplace programming errors such as the omission of brackets, or the use of undefined symbols.

Now that the programmer knows where things are and, to some extent, what (if anything) is wrong, he can go to work debugging his program. Once he locates his errors, he simply pulls the bad cards out of his deck, has new ones punched, and then slips the edited program into a card reader. He may have to repeat this process two or three times before all the bugs are gone, but, once they are, the computer will execute his program with the incredible speed and accuracy that makes this, indeed, the Age of the Computer.



Selected General Bibliography on Computers

ADLER, I. *Thinking Machines.* John Day, New York, 1961.

ARDEN, B. W. *An Introduction to Digital Computing.* Addison-Wesley, Reading, Mass., 1963.

BERKELEY, E. C. *The Computer Revolution.* Doubleday, New York, 1962.

BERNSTEIN, JEREMY. *The Analytical Engine.* Random House, New York, 1963.

ENGLEHART, STANLEY L. *Computers.* Pyramid, New York, 1962.

HOLLINGDALE, S. H., and TOOTILL, G. C. *Electronic Computers.* Penguin Books Ltd., Baltimore, 1965.

LEEDS, HERBERT D., and GERALD M. WEINBERG. *Computer Programming Fundamentals.* McGraw-Hill, New York, 1961.

MCCORMICK, E. M. *Digital Computer Primer.* McGraw-Hill, New York, 1959.

MCCRACKEN, DANIEL D. *A Guide to Fortran.* John Wiley and Sons, New York, 1961.

MORRISON, PHILIP, and MORRISON, EMILY, Ed. *Charles Babbage and His Calculating Engines.* Dover, New York, 1961.

ORGANICK, ELLIOTT I. *A Fortran Primer.* Addison-Wesley, Palo Alto, Calif., 1963.

PFEIFFER, J. P. *The Thinking Machine.* Lippincott, Philadelphia, 1962.

The Authors

D. W. Hagelbarger was born in Kipton, Ohio. He received an A.B. degree from Hiram College in 1942 and a Ph.D. (Physics) from the California Institute of Technology in 1947. From 1946 to 1949, he was Lecturer in Aeronautical Engineering at the University of Michigan. Since 1949, he has been at the Bell Telephone Laboratories where he has done work on special purpose computers, electron dynamics, zone melting, magnetic designs, error correcting codes, and information retrieval. Dr. Hagelbarger is currently a member of the Computing and Information Research Center.

Saul Fingerman joined the Public Relations and Publication Division of the Bell Telephone Laboratories in 1964 after several years at sea as a merchant marine radio officer.

He has written many materials for the Bell System Aids to High School Science Program, including the film, *The Thinking??? Machines*.

A native New Yorker, Mr. Fingerman attended the Bronx High School of Science and, in 1960, received the B.S. degree, magna cum laude, from Columbia University.